Cours SQL

Base du langage SQL et des bases de données

Auteur	Tony Archambeau
Site web	http://sql.sh
Date	24 mai 2014

Sommaire

SQL SELECT....................3
SQL DISTINCT....................6
SQL AS (alias)....................8
SQL WHERE....................11
SQL AND & OR....................13
SQL IN....................15
SQL BETWEEN....................17
SQL LIKE....................19
SQL IS NULL / IS NOT NULL....................21
SQL GROUP BY....................23
SQL HAVING....................25
SQL ORDER BY....................27
SQL LIMIT....................29
SQL CASE....................31
SQL UNION....................35
SQL UNION ALL....................37
SQL INTERSECT....................39
SQL EXCEPT / MINUS....................41
SQL INSERT INTO....................43
SQL ON DUPLICATE KEY UPDATE....................45
SQL UPDATE....................48
SQL DELETE....................49
SQL MERGE....................50
SQL TRUNCATE TABLE....................51
SQL CREATE DATABASE....................52
SQL DROP DATABASE....................53
SQL CREATE TABLE....................54
SQL ALTER TABLE....................56
SQL DROP TABLE....................58
Jointure SQL....................59
SQL INNER JOIN....................60
SQL CROSS JOIN....................62
SQL LEFT JOIN....................64
SQL RIGHT JOIN....................66
SQL FULL JOIN....................68
SQL SELF JOIN....................70
SQL NATURAL JOIN....................72
SQL Sous-requête....................74
SQL EXISTS....................77
SQL ALL....................79
SQL ANY / SOME....................80
Index SQL....................82
SQL CREATE INDEX....................83
SQL EXPLAIN....................85
Commentaires en SQL....................88

SQL SELECT

L'utilisation la plus courante de SQL consiste à lire des données issues de la base de données. Cela s'effectue grâce à la commande SELECT, qui retourne des enregistrements dans un tableau de résultat. Cette commande peut sélectionner une ou plusieurs colonnes d'une table.

Commande basique

L'utilisation basique de cette commande s'effectue de la manière suivante :

```
SELECT nom_du_champ
FROM nom_du_tableau
```

Cette requête va sélectionner (SELECT) le champ « nom_du_champ » provenant (FROM) du tableau appelé « nom_du_tableau ».

Exemple

Imaginons une base de données appelée « client » qui contient des informations sur les clients d'une entreprise.

Table « client » :

identifiant	prenom	nom	ville
1	Pierre	Dupond	Paris
2	Sabrina	Durand	Nantes
3	Julien	Martin	Lyon
4	David	Bernard	Marseille
5	Marie	Leroy	Grenoble

Si l'ont veut avoir la liste de toutes les villes des clients, il suffit d'effectuer la requête suivante :

```
SELECT ville
FROM client
```

Résultat :

ville
Paris
Nantes
Lyon
Marseille
Grenoble

Obtenir plusieurs colonnes

Avec la même table client il est possible de lire plusieurs colonnes à la fois. Il suffit tout simplement de séparer les noms des champs souhaités par une virgule. Pour obtenir les prénoms et les noms des clients il faut alors faire la requête suivante:

```
SELECT prenom, nom
FROM client
```

Résultat :

prenom	nom
Pierre	Dupond
Sabrina	Durand
Julien	Martin
David	Bernard
Marie	Leroy

Obtenir toutes les colonnes d’un tableau

Il est possible de retourner automatiquement toutes les colonnes d’un tableau sans avoir à connaître le nom de toutes les colonnes. Au lieu de lister toutes les colonnes, il faut simplement utiliser le caractère « * » (étoile). C’est un joker qui permet de sélectionner toutes les colonnes. Il s’utilise de la manière suivante :

```
SELECT * FROM client
```

Cette requête retourne exactement les mêmes colonnes qu’il y a dans la base de données. Dans notre cas, le résultat sera donc :

identifiant	prenom	nom	ville
1	Pierre	Dupond	Paris
2	Sabrina	Durand	Nantes
3	Julien	Martin	Lyon
4	David	Bernard	Marseille
5	Marie	Leroy	Grenoble

Il y a des avantages et des inconvénient à l’utiliser. Pour en savoir plus sur le sujet il est recommandé de lire l’article avantage et inconvénient du sélecteur étoile.

Cours avancé : ordre des commandes du SELECT

Cette commande SQL est relativement commune car il est très fréquent de devoir lire les données issues d’une base de données. Il existe plusieurs commandes qui permettent de mieux gérer les données que l’ont souhaite lire. Voici un petit aperçu des fonctionnalités possibles qui sont abordées sur le reste du site:

- Joindre un autre tableau aux résultats
- Filtrer pour ne sélectionner que certains enregistrements
- Classer les résultats
- Grouper les résultats pour faire uniquement des statistiques (note moyenne, prix le plus élevé …)

Un requête SELECT peut devenir assez longue. Juste à titre informatif, voici une requête SELECT qui possède presque toutes les commandes possibles :

```
SELECT *
FROM table
WHERE condition
GROUP BY expression
HAVING condition
{ UNION | INTERSECT | EXCEPT }
ORDER BY expression
LIMIT count
OFFSET start
```

A noter : cette requête imaginaire sert principale d'aide-mémoire pour savoir dans quel ordre sont utilisé chacun des commandes au sein d'une requête SELECT.

SQL DISTINCT

L’utilisation de la commande SELECT en SQL permet de lire toutes les données d’une ou plusieurs colonnes. Cette commande peut potentiellement afficher des lignes en doubles. Pour éviter des redondances dans les résultats il faut simplement ajouter DISTINCT après le mot SELECT.

Commande basique

L’utilisation basique de cette commande consiste alors à effectuer la requête suivante :

```
SELECT DISTINCT ma_colonne
FROM nom_du_tableau
```

Cette requête sélectionne le champ « ma_colonne » de la table « nom_du_tableau » en évitant de retourner des doublons.

Requête pour Oracle

Pour le Système de Gestion de Bases de Données (SGBD) Oracle, cette requête est remplacée par la commande « UNIQUE » :

```
SELECT UNIQUE ma_colonne
FROM nom_du_tableau
```

Exemple

Prenons le cas concret d’une table « client » qui contient des noms et prénoms :

identifiant	prenom	nom
1	Pierre	Dupond
2	Sabrina	Bernard
3	David	Durand
4	Pierre	Leroy
5	Marie	Leroy

En utilisant seulement SELECT tous les noms sont retournés, or la table contient plusieurs fois le même prénom (cf. Pierre). Pour sélectionner uniquement les prénoms uniques il faut utiliser la requête suivante :

```
SELECT DISTINCT prenom
FROM client
```

Résultat :

prenom
Pierre
Sabrina

David
Marie

Ce résultat affiche volontairement qu'une seule fois le prénom « Pierre » grâce à l'utilisation de la commande DISTINCT qui n'affiche que les résultats distincts.

Intérêt

L'utilisation de la commande DISTINCT est très pratique pour éviter les résultats en doubles. Cependant, pour optimiser les performances il est préférable d'utiliser la commande SQL GROUP BY lorsque c'est possible.

SQL AS (alias)

Dans le langage SQL il est possible d'utiliser des alias pour renommer temporairement une colonne ou une table dans une requête. Cette astuce est particulièrement utile pour faciliter la lecture des requêtes.

Intérêts et utilités

Alias sur une colonne

Permet de renommer le nom d'une colonne dans les résultats d'une requête SQL. C'est pratique pour avoir un nom facilement identifiable dans une application qui doit ensuite exploiter les résultats d'une recherche.

Cas concrets d'utilisations :

- Une colonne qui s'appelle normalement c_iso_3166 peut être renommée « code_pays » (cf. le code ISO 3166 correspond au code des pays), ce qui est plus simple à comprendre dans le reste du code par un développeur.
- Une requête qui utilise la commande UNION sur des champs aux noms différents peut être ambigu pour un développeur. En renommant les champs avec un même nom il est plus simple de traiter les résultats.
- Lorsqu'une fonction est utilisé, le nom d'une colonne peut-être un peu complexe. Il est ainsi possible de renommer la colonne sur laquelle il y a une fonction SQL. Exemple : SELECT COUNT(*) AS nombre_de_resultats FROM `table`.
- Lorsque plusieurs colonnes sont combinées il est plus simple de renommer la nouvelle colonne qui est une concaténation de plusieurs champs.

Alias sur une table

Permet d'attribuer un autre nom à une table dans une requête SQL. Cela peut aider à avoir des noms plus court, plus simple et plus facilement compréhensible. Ceci est particulièrement vrai lorsqu'il y a des jointures.

Syntaxe

Alias sur une colonne

La syntaxe pour renommer une colonne de colonne1 à c1 est la suivante :

```
SELECT colonne1 AS c1, colonne2
FROM `table`
```

Cette syntaxe peut également s'afficher de la façon suivante :

```
SELECT colonne1 c1, colonne2
FROM `table`
```

A noter : à choisir il est préférable d'utiliser la commande « AS » pour que ce soit plus explicite (plus simple à lire qu'un simple espace), d'autant plus que c'est recommandé dans le standard ISO pour concevoir une requête SQL.

Alias sur une table

La syntaxe pour renommer une table dans une requête est la suivante :

```
SELECT *
FROM `nom_table` AS t1
```

Cette requête peut également s'écrire de la façon suivante :

```
SELECT *
FROM `table`t1
```

Exemple

Renommer une colonne

Imaginons une site d'e-commerce qui possède une table de produits. Ces produits sont disponibles dans une même table dans plusieurs langues , dont le français. Le nom du produit peut ainsi être disponible dans la colonne « nom_fr_fr », « nom_en_gb » ou « nom_en_us ». Pour utiliser l'un ou l'autre des titres dans le reste de l'application sans avoir à se soucier du nom de la colonne, il est possible de renommer la colonne de son choix avec un nom générique. Dans notre cas, la requête pourra ressemble à ceci :

```
SELECT p_id, p_nom_fr_fr AS nom, p_description_fr_fr AS description,
p_prix_euro AS prix
FROM `produit`
```

Résultat :

id	nom	description	prix
1	Ecran	Ecran de grandes tailles.	399.99
2	Clavier	Clavier sans fil.	27
3	Souris	Souris sans fil.	24
4	Ordinateur portable	Grande autonomie et et sacoche offerte.	700

Comme nous pouvons le constater les colonnes ont été renommées.

Renommer une ou plusieurs tables

Imaginons que les produits du site e-commerce soit répartis dans des catégories. Pour récupérer la liste des produits en même temps que la catégorie auquel il appartient il est possible d'utiliser une requête SQL avec une jointure. Cette requête peut utiliser des alias pour éviter d'utiliser à chaque fois le nom des tables.

La requête ci-dessous renomme la table « produit » en « p » et la table « produit_categorie » en « pc » (plus court et donc plus rapide à écrire) :

```
SELECT p_id, p_nom_fr_fr, pc_id, pc_nom_fr_fr
FROM `produit` AS p
LEFT JOIN `produit_categorie` AS pc ON pc.pc_id = p.p_fk_category_id
```

Cette astuce est encore plus pratique lorsqu'il y a des noms de tables encore plus compliqués et lorsqu'il y a beaucoup de jointures.

SQL WHERE

La commande WHERE dans une requête SQL permet d'extraire les lignes d'une base de données qui respectent une condition. Cela permet d'obtenir uniquement les informations désirées.

Syntaxe

La commande WHERE s'utilise en complément à une requête utilisant SELECT. La façon la plus simple de l'utiliser est la suivante :

```
SELECT nom_colonnes
FROM nom_table
WHERE condition
```

Exemple

Imaginons une base de données appelée « client » qui contient le nom des clients, le nombre de commandes qu'ils ont effectués et leur ville :

id	nom	nbr_commande	ville
1	Paul	3	paris
2	Maurice	0	rennes
3	Joséphine	1	toulouse
4	Gérard	7	paris

Pour obtenir seulement la liste des clients qui habitent à Paris, il faut effectuer la requête suivante :

```
SELECT *
FROM client
WHERE ville = 'paris'
```

Résultat :

id	nom	nbr_commande	nbr_commande
1	Paul	3	paris
4	Gérard	7	paris

Attention : dans notre cas tout est en minuscule donc il n'y a pas eu de problème. Cependant, si un table est sensible à la casse, il faut faire attention aux majuscules et minuscules.

Opérateurs de comparaisons

Il existe plusieurs opérateurs de comparaisons. La liste ci-jointe présente quelques uns des opérateurs les plus couramment utilisés.

Opérateur	Description
=	Égale
<>	Pas égale
!=	Pas égale
>	Supérieur à
<	Inférieur à
>=	Supérieur ou égale à
<=	Inférieur ou égale à
IN	Liste de plusieurs valeurs possibles
BETWEEN	Valeur comprise dans un intervalle donnée (utile pour les nombres ou dates)
LIKE	Recherche en spécifiant le début, milieu ou fin d'un mot.
IS NULL	Valeur est nulle
IS NOT NULL	Valeur n'est pas nulle

Attention : il y a quelques opérateurs qui n’existe pas dans des vieilles versions de système de gestion de bases de données (SGBD). De plus, il y a de nouveaux opérateurs non indiqués ici qui sont disponibles avec certains SGBD. N’hésitez pas à consulter la documentation de MySQL, PostgreSQL ou autre pour voir ce qu’il vous est possible de faire.

SQL AND & OR

Une requête SQL peut être restreinte à l'aide de la condition WHERE. Les opérateurs logiques AND et OR peuvent être utilisées au sein de la commande WHERE pour combiner des conditions.

Syntaxe d'utilisation des opérateurs AND et OR

Les opérateurs sont à ajoutés dans la condition WHERE. Ils peuvent être combinés à l'infini pour filtrer les données comme souhaités.

L'opérateur AND permet de s'assurer que la condition1 ET la condition2 sont vrai :

```
SELECT nom_colonnes
FROM nom_table
WHERE condition1 AND condition2
```

L'opérateur OR vérifie quant à lui que la condition1 OU la condition2 est vrai :

```
SELECT nom_colonnes FROM nom_table
 WHERE condition1 OR condition2
```

Ces opérateurs peuvent être combinés à l'infini et mélangés. L'exemple ci-dessous filtre les résultats de la table « nom_table » si condition1 ET condition2 OU condition3 est vrai :

```
SELECT nom_colonnes FROM nom_table
 WHERE condition1 AND (condition2 OR condition3)
```

Attention : il faut penser à utiliser des parenthèses lorsque c'est nécessaire. Cela permet d'éviter les erreurs car et ça améliore la lecture d'une requête par un humain.

Exemple de données

Pour illustrer les prochaines commandes, nous allons considérer la table « produit » suivante :

id	nom	categorie	stock	prix
1	ordinateur	informatique	5	950
2	clavier	informatique	32	35
3	souris	informatique	16	30
4	crayon	fourniture	147	2

Opérateur AND

L'opérateur AND permet de joindre plusieurs conditions dans une requête. En gardant la même table que précédemment, pour filtrer uniquement les produits informatique qui sont presque en rupture de stock (moins de 20 produits disponible) il faut exécuter la requête suivante :

```
SELECT * FROM produit
WHERE categorie = 'informatique' AND stock < 20
```

Résultat :

id	nom	categorie	stock	prix
1	ordinateur	informatique	5	950
3	souris	informatique	16	30

Opérateur OR

Pour filtrer les données pour avoir uniquement les données sur les produits « ordinateur » ou « clavier » il faut effectuer la recherche suivante :

```
SELECT * FROM produit
WHERE nom = 'ordinateur' OR nom = 'clavier'
```

Résultats :

id	nom	categorie	stock	prix
1	ordinateur	informatique	5	950
2	clavier	informatique	32	35

Combiner AND et OR

Il ne faut pas oublier que les opérateurs peuvent être combinés pour effectuer de puissantes recherche. Il est possible de filtrer les produits « informatique » avec un stock inférieur à 20 et les produits « fourniture » avec un stock inférieur à 200 avec la recherche suivante :

```
SELECT * FROM produit
WHERE ( categorie = 'informatique' AND stock < 20 )
OR ( categorie = 'fourniture' AND stock < 200 )
```

Résultats :

id	nom	categorie	stock	prix
1	ordinateur	informatique	5	950
2	clavier	informatique	32	35
4	crayon	fourniture	147	2

SQL IN

L’opérateur logique IN dans SQL s’utilise avec la commande WHERE pour vérifier si une colonne est égale à une des valeurs comprise dans set de valeurs déterminés. C’est une méthode simple pour vérifier si une colonne est égale à une valeur OU une autre valeur OU une autre valeur et ainsi de suite, sans avoir à utiliser de multiple fois l’opérateur OR.

Syntaxe

Pour chercher toutes les lignes où la colonne « nom_colonne » est égale à ‘valeur 1′ OU ‘valeur 2′ ou ‘valeur 3′, il est possible d’utiliser la syntaxe suivante :

```
SELECT nom_colonne
FROM table
WHERE nom_colonne IN ( valeur1, valeur2, valeur3, ... )
```

A savoir : entre les parenthèses il n’y a pas de limite du nombre d’arguments. Il est possible d’ajouter encore d’autres valeurs.

Cette syntaxe peut être associée à l’opérateur NOT pour recherche toutes les lignes qui ne sont pas égales à l’une des valeurs stipulées.

Simplicité de l’opérateur IN

La syntaxe utilisée avec l’opérateur est plus simple que d’utiliser une succession d’opérateur OR. Pour le montrer concrètement avec un exemple, voici 2 requêtes qui retournerons les mêmes résultats, l’une utilise l’opérateur IN, tandis que l’autre utilise plusieurs OR.

Requête avec plusieurs OR

```
SELECT prenom
FROM utilisateur
WHERE prenom = 'Maurice' OR prenom = 'Marie' OR prenom = 'Thimoté'
```

Requête équivalent avec l’opérateur IN

```
SELECT prenom
FROM utilisateur
WHERE prenom IN ( 'Maurice', 'Marie', 'Thimoté' )
```

Exemple

Imaginons une table « adresse » qui contient une liste d’adresse associée à des utilisateurs d’une application.

id	id_utilisateur	addr_rue	addr_code_postal	addr_ville
1	23	35 Rue Madeleine Pelletier	25250	Bournois
2	43	21 Rue du Moulin Collet	75006	Paris

3	65	28 Avenue de Cornouaille	27220	Mousseaux-Neuville
4	67	41 Rue Marcel de la Provoté	76430	Graimbouville
5	68	18 Avenue de Navarre	75009	Paris

Si l'ont souhaite obtenir les enregistrements des adresses de Paris et de Graimbouville, il est possible d'utiliser la requête suivante :

```
SELECT *
FROM adresse
WHERE addr_ville IN ( 'Paris', 'Graimbouville' )
```

Résultats :

id	id_utilisateur	addr_rue	addr_code_postal	addr_ville
2	43	21 Rue du Moulin Collet	75006	Paris
4	67	41 Rue Marcel de la Provoté	76430	Graimbouville
5	68	18 Avenue de Navarre	75009	Paris

SQL BETWEEN

L’opérateur BETWEEN est utilisé dans une requête SQL pour sélectionner un intervalle de données dans une requête utilisant WHERE. L’intervalle peut être constitué de chaînes de caractères, de nombres ou de dates. L’exemple le plus concret consiste par exemple à récupérer uniquement les enregistrements entre 2 dates définies.

Syntaxe

L’utilisation de la commande BETWEEN s’effectue de la manière suivante :

```
SELECT *
FROM table
WHERE nom_colonne BETWEEN 'valeur1' AND 'valeur2'
```

La requête suivante retournera toutes les lignes dont la valeur de la colonne « nom_colonne » sera comprise entre valeur1 et valeur2.

Exemple : filtrer entre 2 dates

Imaginons une table « utilisateur » qui contient les membres d’une application en ligne.

id	nom	date_inscription
1	Maurice	2012-03-02
2	Simon	2012-03-05
3	Chloé	2012-04-14
4	Marie	2012-04-15
5	Clémentine	2012-04-26

Si l’ont souhaite obtenir les membres qui se sont inscrit entre le 1 avril 2012 et le 20 avril 2012 il est possible d’effectuer la requête suivante :

```
SELECT *
FROM utilisateur
WHERE date_inscription BETWEEN ’2012-04-01' AND ’2012-04-20'
```

Résultat :

id	nom	date_inscription
3	Chloé	2012-04-14
4	Marie	2012-04-15

Exemple : filtrer entre 2 entiers

Si l’ont souhaite obtenir tous les résultats dont l’identifiant n’est pas situé entre 4 et 10, il faudra alors utiliser la requête suivante :

```
SELECT *
FROM utilisateur
WHERE id NOT BETWEEN 4 AND 10
```

Résultat :

id	nom	date_inscription
1	Maurice	2012-03-02
2	Simon	2012-03-05
3	Chloé	2012-04-14

Bon à savoir

Certaines vieilles versions de systèmes de gestion de bases de données ne prennent pas en compte la commande BETWEEN. Mais si vous utilisez une version récente de MySQL ou PostgreSQL, cela ne cause aucun problème.

L'autre élément important à savoir c'est que toutes les bases de données ne gèrent pas l'opérateur BETWEEN de la même manière. Certains systèmes vont inclurent les valeurs qui définissent l'intervalle tandis que d'autres systèmes considèrent ces valeurs sont exclues. Il est important de consulter la documentation officielle de la base de données que vous utilisez pour avoir une réponse exacte à ce sujet.

SQL LIKE

L'opérateur LIKE est utilisé dans la clause WHERE des requêtes SQL. Ce mot-clé permet d'effectuer une recherche sur un modèle particulier. Il est par exemple possible de rechercher les enregistrements dont la valeur d'une colonne commence par telle ou telle lettre. Les modèles de recherches sont multiple.

Syntaxe

La syntaxe à utiliser pour utiliser l'opérateur LIKE est la suivante :

```
SELECT *
FROM table
WHERE colonne LIKE modele
```

Dans cet exemple le « modèle » n'a pas été défini, mais il ressemble très généralement à l'un des exemples suivants :

- **LIKE '%a' :** le caractère « % » est un caractère joker qui remplace tous les autres caractères. Ainsi, ce modèle permet de rechercher toutes les chaines de caractère qui se termine par un « a ».
- **LIKE 'a%' :** ce modèle permet de rechercher toutes les lignes de « colonne » qui commence par un « a ».
- **LIKE '%a%' :** ce modèle est utilisé pour rechercher tous les enregistrement qui utilisent le caractère « a ».
- **LIKE 'pa%on' :** ce modèle permet de rechercher les chaines qui commence par « pa » et qui se terminent par « on », comme « pantalon » ou « pardon ».
- **LIKE 'a_c' :** peu utilisé, le caractère « _ » (underscore) peut être remplacé par n'importe quel caractère, mais un seul caractère uniquement (alors que le symbole pourcentage « % » peut être remplacé par un nombre incalculable de caractères . Ainsi, ce modèle permet de retourner les lignes « aac », « abc » ou même « azc ».

Exemple

Imaginons une table « client » qui contient les enregistrement d'utilisateurs :

id	nom	ville
1	Léon	Lyon
2	Odette	Nice
3	Vivien	Nantes
4	Etienne	Lille

Obtenir les résultats qui commencent par « N »

Si l'ont souhaite obtenir uniquement les clients des villes qui commencent par un « N », il est possible d'utiliser la requête suivante :

```
SELECT *
FROM client
WHERE ville LIKE 'N%'
```

Avec cette requête, seul les enregistrements suivants seront retournés :

id	nom	ville
2	Odette	Nice
3	Vivien	Nantes

Obtenir les résultats terminent par « e »

Requête :

```
SELECT *
FROM client
WHERE ville LIKE '%e'
```

Résultat :

id	nom	ville
2	Odette	Nice
4	Etienne	Lille

SQL IS NULL / IS NOT NULL

Dans le langage SQL, l'opérateur IS permet de filtrer les résultats qui contiennent la valeur NULL. Cet opérateur est indispensable car la valeur NULL est une valeur inconnue et ne peut par conséquent pas être filtrée par les opérateurs de comparaison (cf. égal, inférieur, supérieur ou différent).

Syntaxe

Pour filtrer les résultats où les champs d'une colonne sont à NULL il convient d'utiliser la syntaxe suivante :

```
SELECT *
FROM `table`
WHERE nom_colonne IS NULL
```

A l'inverse, pour filtrer les résultats et obtenir uniquement les enregistrements qui ne sont pas null, il convient d'utiliser la syntaxe suivante :

```
SELECT *
FROM `table`
WHERE nom_colonne IS NOT NULL
```

A savoir : l'opérateur IS retourne en réalité un booléen, c'est à dire une valeur TRUE si la condition est vrai ou FALSE si la condition n'est pas respectée. Cet opérateur est souvent utilisé avec la condition WHERE mais peut aussi trouvé son utilité lorsqu'une sous-requête est utilisée.

Exemple

Imaginons une application qui possède une table contenant les utilisateurs. Cette table possède 2 colonnes pour associer les adresses de livraison et de facturation à un utilisateur (grâce à une clé étrangère). Si cet utilisateur n'a pas d'adresse de facturation ou de livraison, alors le champ reste à NULL.

Table « utilisateur » :

id	nom	date_inscription	fk_adresse_livraison_id	fk_adresse_facturation_id
23	Grégoire	2013-02-12	12	12
24	Sarah	2013-02-17	NULL	NULL
25	Anne	2013-02-21	13	14
26	Frédérique	2013-03-02	NULL	NULL

Exemple 1 : utilisateurs sans adresse de livraison

Il est possible d'obtenir la liste des utilisateurs qui ne possèdent pas d'adresse de livraison en utilisant la requête SQL suivante :

```
SELECT *
FROM `utilisateur`
WHERE `fk_adresse_livraison_id` IS NULL
```

Résultat :

id	nom	date_inscription	fk_adresse_livraison_id	fk_adresse_facturation_id
24	Sarah	2013-02-17	NULL	NULL
26	Frédérique	2013-03-02	NULL	NULL

Les enregistrements retournés montrent bien que seul les utilisateurs ayant la valeur NULL pour le champ de l'adresse de livraison.

Exemple 2 : utilisateurs avec une adresse de livraison

Pour obtenir uniquement les utilisateurs qui possèdent une adresse de livraison il convient de lancer la requête SQL suivante :

```
SELECT *
FROM `utilisateur`
WHERE `fk_adresse_livraison_id` IS NOT NULL
```

Résultat :

id	nom	date_inscription	fk_adresse_livraison_id	fk_adresse_facturation_id
23	Grégoire	2013-02-12	12	12
25	Anne	2013-02-21	13	14

Les lignes retournés sont exclusivement celles qui n'ont pas une valeur NULL pour le champ de l'adresse de livraison.

SQL GROUP BY

La commande GROUP BY est utilisée en SQL pour grouper plusieurs résultats et utiliser une fonction de totaux sur un groupe de résultat. Sur une table qui contient toutes les ventes d'un magasin, il est par exemple possible de liste regrouper les ventes par clients identiques et d'obtenir le coût total des achats pour chaque client.

Syntaxe d'utilisation de GROUP BY

De façon générale, la commande GROUP BY s'utilise de la façon suivante :

```
SELECT colonne1, fonction(colonne2)
FROM table
GROUP BY colonne1
```

A noter : cette commande doit toujours s'utiliser après la commande WHERE et avant la commande HAVING.

Exemple d'utilisation

Prenons en considération une table « achat » qui résume les ventes d'une boutique :

id	client	tarif	date
1	Pierre	102	2012-10-23
2	Simon	47	2012-10-27
3	Marie	18	2012-11-05
4	Marie	20	2012-11-14
5	Pierre	160	2012-12-03

Ce tableau contient une colonne qui sert d'identifiant pour chaque ligne, une autre qui contient le nom du client, le coût de la vente et la date d'achat.

Pour obtenir le coût total de chaque client en regroupant les commandes des mêmes clients, il faut utiliser la requête suivante :

```
SELECT client, SUM(tarif)
FROM achat
GROUP BY client
```

La fonction SUM() permet d'additionner la valeur de chaque tarif pour un même client. Le résultat sera donc le suivant :

client	SUM(tarif)
Pierre	262
Simon	47
Marie	38

La manière simple de comprendre le GROUP BY c'est tout simplement d'assimiler qu'il va éviter de présenter plusieurs fois les mêmes lignes. C'est une méthode pour éviter les doublons.

Juste à titre informatif, voici ce qu'on obtient de la requête sans utiliser GROUP BY.

Requête :

```
SELECT client, SUM(tarif)
FROM achat
```

Résultat :

client	SUM(tarif)
Pierre	262
Simon	47
Marie	38
Marie	38
Pierre	262

Utilisation d'autres fonctions de statistiques

Il existe plusieurs fonctions qui peuvent être utilisées pour manipuler plusieurs enregistrements, il s'agit des fonctions d'agrégations statistiques, les principales sont les suivantes :

- AVG() pour calculer la moyenne d'un set de valeur. Permet de connaître le prix du panier moyen pour de chaque client
- COUNT() pour compter le nombre de lignes concernées. Permet de savoir combien d'achats a été effectué par chaque client
- MAX() pour récupérer la plus haute valeur. Pratique pour savoir l'achat le plus cher
- MIN() pour récupérer la plus petite valeur. Utile par exemple pour connaître la date du premier achat d'un client
- SUM() pour calculer la somme de plusieurs lignes. Permet par exemple de connaître le total de tous les achats d'un client

Ces petites fonctions se révèlent rapidement indispensable pour travailler sur des données.

SQL HAVING

La condition HAVING en SQL est presque similaire à WHERE à la seule différence que HAVING permet de filtrer en utilisant des fonctions telles que SUM(), COUNT(), AVG(), MIN() ou MAX().

Syntaxe

L'utilisation de HAVING s'utilise de la manière suivante :

```
SELECT colonne1, SUM(colonne2)
FROM nom_table
GROUP BY colonne1
HAVING fonction(colonne2) operateur valeur
```

Cela permet donc de SÉLECTIONNER les colonnes DE la table « nom_table » en GROUPANT les lignes qui ont des valeurs identiques sur la colonne « colonne1″ et que la condition de HAVING soit respectée.

Important : HAVING est très souvent utilisé en même temps que GROUP BY bien que ce ne soit pas obligatoire.

Exemple

Pour utiliser un exemple concret, imaginons une table « achat » qui contient les achats de différents clients avec le coût du panier pour chaque achat.

id	client	tarif	date_achat
1	Pierre	102	2012-10-23
2	Simon	47	2012-10-27
3	Marie	18	2012-11-05
4	Marie	20	2012-11-14
5	Pierre	160	2012-12-03

Si dans cette table on souhaite récupérer la liste des clients qui ont commandé plus de 40€, toute commandes confondu alors il est possible d'utiliser la requête suivante :

```
SELECT client, SUM(tarif)
FROM achat
GROUP BY client
HAVING SUM(tarif) > 40
```

Résultat :

client	SUM(tarif)
Pierre	162
Simon	47

La cliente « Marie » a cumulée 38€ d’achat (un achat de 18€ et un autre de 20€) ce qui est inférieur à la limite de 40€ imposée par HAVING. En conséquent cette ligne n’est pas affichée dans le résultat.

SQL ORDER BY

La commande ORDER BY permet de trier les lignes dans un résultat d'une requête SQL. Il est possible de trier les données sur une ou plusieurs colonnes, par ordre ascendant ou descendant.

Syntaxe

Une requête où l'ont souhaite filtrer l'ordre des résultats utilise la commande ORDER BY de la sorte :

```
SELECT colonne1, colonne2
FROM table
ORDER BY colonne1
```

Par défaut les résultats sont classés par ordre ascendant, toutefois il est possible d'inverser l'ordre en utilisant le suffixe DESC après le nom de la colonne. Par ailleurs, il est possible de trier sur plusieurs colonnes en les séparant par une virgule. Une requête plus élaboré ressemblerais alors cela :

```
SELECT colonne1, colonne2, colonne3
FROM table
ORDER BY colonne1 DESC, colonne2 ASC
```

A noter : il n'est pas obligé d'utiliser le suffixe « ASC » sachant que les résultats sont toujours classé par ordre ascendant par défaut. Toutefois, c'est plus pratique pour mieux s'y retrouver, surtout si on a oublié l'ordre par défaut.

Exemple

Pour l'ensemble de nos exemple, nous allons prendre un base « utilisateur » de test, qui contient les données suivantes :

id	nom	prenom	date_inscription	tarif_total
1	Durand	Maurice	2012-02-05	145
2	Dupond	Fabrice	2012-02-07	65
3	Durand	Fabienne	2012-02-13	90
4	Dubois	Chloé	2012-02-16	98
5	Dubois	Simon	2012-02-23	27

Pour récupérer la liste de ces utilisateurs par ordre alphabétique du nom de famille, il est possible d'utiliser la requête suivante :

```
SELECT *
FROM utilisateur
ORDER BY nom
```

Résultat :

id	nom	prenom	date_inscription	tarif_total
4	Dubois	Chloé	2012-02-16	98
5	Dubois	Simon	2012-02-23	27
2	Dupond	Fabrice	2012-02-07	65
1	Durand	Maurice	2012-02-05	145
3	Durand	Fabienne	2012-02-13	90

En utilisant deux méthodes de tri, il est possible de retourner les utilisateurs par ordre alphabétique ET pour ceux qui ont le même nom de famille, les trier par ordre décroissant d'inscription. La requête serait alors la suivante :

```
SELECT *
FROM utilisateur
ORDER BY nom, date_inscription DESC
```

Résultat :

id	nom	prenom	date_inscription	tarif_total
5	Dubois	Simon	2012-02-23	27
4	Dubois	Chloé	2012-02-16	98
2	Dupond	Fabrice	2012-02-07	65
3	Durand	Fabienne	2012-02-13	90
1	Durand	Maurice	2012-02-05	145

SQL LIMIT

La clause LIMIT est à utiliser dans une requête SQL pour spécifier le nombre maximum de résultats que l'ont souhaite obtenir. Cette clause est souvent associé à un OFFSET, c'est-à-dire effectuer un décalage sur le jeu de résultat. Ces 2 clauses permettent par exemple d'effectuer des système de pagination (exemple : récupérer les 10 articles de la page 4).

ATTENTION : selon le système de gestion de base de données, la syntaxe ne sera pas pareil. Ce tutoriel va donc présenter la syntaxe pour MySQL et pour PostgreSQL.

Syntaxe simple

La syntaxe commune aux principales système de gestion de bases de données est la suivante :

```
SELECT *
FROM table
LIMIT 10
```

Cette requête permet de récupérer seulement les 10 premiers résultats d'une table. Bien entendu, si la table contient moins de 10 résultats, alors la requête retournera toutes les lignes.

Bon à savoir : la bonne pratique lorsque l'ont utilise LIMIT consiste à utiliser également la clause ORDER BY pour s'assurer que quoi qu'il en soit ce sont toujours les bonnes données qui sont présentées. En effet, si le système de tri est non spécifié, alors il est en principe inconnu et les résultats peuvent être imprévisible.

Limit et Offset avec PostgreSQL

L'offset est une méthode simple de décaler les lignes à obtenir. La syntaxe pour utiliser une limite et un offset est la suivante :

```
SELECT *
FROM table
LIMIT 10 OFFSET 5
```

Cette requête permet de récupérer les résultats 6 à 15 (car l'OFFSET commence toujours à 0). A titre d'exemple, pour récupérer les résultats 16 à 25 il faudrait donc utiliser: LIMIT 10 OFFSET 15

A noter : Utiliser OFFSET 0 reviens au même que d'omettre l'OFFSET.

Limit et Offset avec MySQL

La syntaxe avec MySQL est légèrement différente :

```
SELECT *
FROM table
LIMIT 5, 10;
```

Cette requête retourne les enregistrements 6 à 15 d'une table. Le premier nombre est l'OFFSET tandis que le suivant est la limite.

Bon à savoir : pour une bonne compatibilité, MySQL accepte également la syntaxe LIMIT nombre OFFSET nombre. En conséquent, dans la conception d'une application utilisant MySQL il est préférable d'utiliser cette syntaxe car c'est potentiellement plus facile de migrer vers un autre système de gestion de base de données sans avoir à ré-écrire toutes les requêtes.

Performance

Ce dernier chapitre est destiné à un public averti. Il n'est pas nécessaire de le comprendre entièrement, mais simplement d'avoir compris les grandes lignes.

Certains développeur pensent à tort que l'utilisation de LIMIT permet de réduire le temps d'exécution d'une requête. Or, le temps d'exécution est sensiblement le même car la requête va permettre de récupérer toutes les lignes (donc temps d'exécution identique) PUIS seulement les résultats définit par LIMIT et OFFSET seront retournés. Au mieux, utiliser LIMIT permet de réduire le temps d'affichage car il y a moins de lignes à afficher.

SQL CASE

Dans le langage SQL, la commande « CASE … WHEN … » permet d’utiliser des conditions de type « si / sinon » (cf. if / else) similaire à un langage de programmation pour retourner un résultat disponible entre plusieurs possibilités. Le CASE peut être utilisé dans n’importe quelle instruction ou clause, telle que SELECT, UPDATE, DELETE, WHERE, ORDER BY ou HAVING.

Syntaxe

L’utilisation du CASE est possible de 2 manières différentes :

- Comparer une colonne à un set de résultat possible
- Élaborer une série de conditions booléennes pour déterminer un résultat

Comparer une colonne à un set de résultat

Voici la syntaxe nécessaire pour comparer une colonne à un set d’enregistrement :

```
CASE a
       WHEN 1 THEN 'un'
       WHEN 2 THEN 'deux'
       WHEN 3 THEN 'trois'
       ELSE 'autre'
END
```

Dans cet exemple les valeurs contenus dans la colonne « a » sont comparé à 1, 2 ou 3. Si la condition est vrai, alors la valeur située après le THEN sera retournée.

A noter : la condition ELSE est facultative et sert de ramasse-miette. Si les conditions précédentes ne sont pas respectées alors ce sera la valeur du ELSE qui sera retournée par défaut.

Élaborer une série de conditions booléennes pour déterminer un résultat

Il est possible d’établir des conditions plus complexes pour récupérer un résultat ou un autre. Cela s’effectue en utilisant la syntaxe suivante :

```
CASE
     WHEN a=b THEN 'A égal à B'
     WHEN a>b THEN 'A supérieur à B'
     ELSE 'A inférieur à B'
END
```

Dans cet exemple les colonnes « a », « b » et « c » peuvent contenir des valeurs numériques. Lorsqu’elles sont respectées, les conditions booléennes permettent de rentrer dans l’une ou l’autre des conditions.

Il est possible de reproduire le premier exemple présenté sur cette page en utilisant la syntaxe suivante :

```
CASE
       WHEN a=1 THEN 'un'
```

```
        WHEN a=2 THEN 'deux'
        WHEN a=3 THEN 'trois'
        ELSE 'autre'
END
```

Exemple

Pour présenter le CASE dans le langage SQL il est possible d'imaginer une base de données utilisées par un site de vente en ligne. Dans cette base il y a une table contenant les achats, cette table contient le nom des produits, le prix unitaire, la quantité achetée et une colonne consacrée à une marge fictive sur certains produits.

Table « achat » :

id	nom	surcharge	prix_unitaire	quantite
1	Produit A	1.3	6	3
2	Produit B	1.5	8	2
3	Produit C	0.75	7	4
4	Produit D	1	15	2

Afficher un message selon une condition

Il est possible d'effectuer une requête qui va afficher un message personnalisé en fonction de la valeur de la marge. Le message sera différent selon que la marge soit égale à 1, supérieur à 1 ou inférieure à 1. La requête peut se présenter de la façon suivante :

```
SELECT id, nom, marge_pourcentage, prix_unitaire, quantite,
    CASE
      WHEN marge_pourcentage=1 THEN 'Prix ordinaire'
      WHEN marge_pourcentage>1 THEN 'Prix supérieur à la normale'
      ELSE 'Prix inférieur à la normale'
    END
FROM `achat`
```

Résultat :

id	nom	surcharge	prix_unitaire	quantite	CASE
1	Produit A	1.3	6	3	Prix supérieur à la normale
2	Produit B	1.5	8	2	Prix supérieur à la normale
3	Produit C	0.75	7	4	Prix inférieur à la normale
4	Produit D	1	15	2	Prix ordinaire

Ce résultat montre qu'il est possible d'afficher facilement des messages personnalisés selon des conditions simples.

Afficher un prix unitaire différent selon une condition

Avec un CASE il est aussi possible d'utiliser des requêtes plus élaborées. Imaginons maintenant que nous souhaitions multiplier le prix unitaire par 2 si la marge est supérieur à 1, la diviser par 2 si la marge est inférieure à 1 et laisser le prix unitaire tel quel si la marge est égale à 1. C'est possible grâce à la requête SQL :

```
SELECT id, nom, marge_pourcentage, prix_unitaire, quantite,
  CASE
    WHEN marge_pourcentage=1 THEN prix_unitaire
    WHEN marge_pourcentage>1 THEN prix_unitaire*2
    ELSE prix_unitaire/2
   END
FROM `achat`
```

Résultat :

id	nom	surcharge	prix_unitaire	quantite	CASE
1	Produit A	1.3	6	3	12
2	Produit B	1.5	8	2	16
3	Produit C	0.75	7	4	3.5
4	Produit D	1	15	2	15

Comparer un champ à une valeur donnée

Imaginons maintenant que l'application propose des réductions selon le nombre de produits achetés :

- 1 produit acheté permet d'obtenir une réduction de -5% pour le prochain achat
- 2 produit acheté permet d'obtenir une réduction de -6% pour le prochain achat
- 3 produit acheté permet d'obtenir une réduction de -8% pour le prochain achat
- Pour plus de produits achetés il y a un réduction de -10% pour le prochain achat

Pour effectuer une telle procédure, il est possible de comparer la colonne « quantite » aux différentes valeurs spécifiée et d'afficher un message personnalisé en fonction du résultat. Cela peut être réalisé avec cette requête SQL :

```
SELECT id, nom, marge_pourcentage, prix_unitaire, quantite,
  CASE quantite
    WHEN 0 THEN 'Erreur'
    WHEN 1 THEN 'Offre de -5% pour le prochain achat'
    WHEN 2 THEN 'Offre de -6% pour le prochain achat'
    WHEN 3 THEN 'Offre de -8% pour le prochain achat'
    ELSE 'Offre de -10% pour le prochain achat'
  END
 FROM `achat`
```

Résultat :

id	nom	surcharge	prix_unitaire	quantite	CASE
1	Produit A	1.3	6	3	Offre de -8% pour le prochain achat
2	Produit B	1.5	8	2	Offre de -6% pour le prochain achat
3	Produit C	0.75	7	4	Offre de -10% pour le prochain achat
4	Produit D	1	15	2	Offre de -6% pour le prochain achat

Astuce : la condition ELSE peut parfois être utilisée pour gérer les erreurs.

UPDATE avec CASE

Comme cela a été expliqué au début, il est aussi possible d’utiliser le CASE à la suite de la commande SET d’un UPDATE pour mettre à jour une colonne avec une données spécifique selon une règle. Imaginons par exemple que l’ont souhaite offrir un produit pour tous les achats qui ont une surcharge inférieur à 1 et que l’ont souhaite retirer un produit pour tous les achats avec une surcharge supérieur à 1. Il est possible d’utiliser la requête SQL suivante :

```
UPDATE `achat`
SET `quantite` = (
  CASE
    WHEN `surcharge` < 1 THEN `quantite` + 1
    WHEN `surcharge` > 1 THEN `quantite` - 1
    ELSE quantite
  END
)
```

SQL UNION

La commande UNION de SQL permet de mettre bout-à-bout les résultats de plusieurs requêtes utilisant elles-même la commande SELECT. C'est donc une commande qui permet de concaténer les résultats de 2 requêtes ou plus. Pour l'utiliser il est nécessaire que chacune des requêtes à concaténer retournes le même nombre de colonnes, avec les mêmes types de données et dans le même ordre.

A savoir : par défaut, les enregistrements exactement identiques ne seront pas répétés dans les résultats. Pour effectuer une union dans laquelle même les lignes dupliquées sont affichées il faut plutôt utiliser la commande UNION ALL.

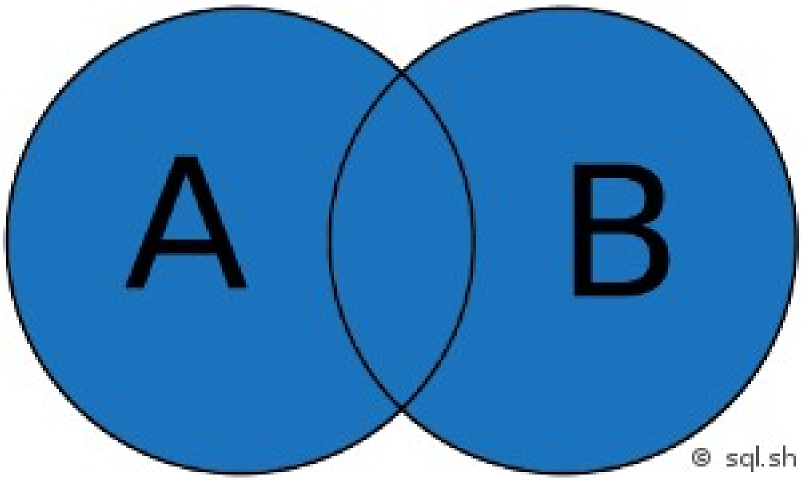
Syntaxe

La syntaxe pour unir les résultats de 2 tableaux sans afficher les doublons est la suivante :

```
SELECT * FROM table1
UNION
SELECT * FROM table2
```

Schéma explicatif

L'union de 2 ensembles A et B est un concept qui consiste à obtenir tous les éléments qui correspondent à la fois à l'ensemble A ou à l'ensemble B. Cela se résume très simplement par un petit schéma où la zone en bleu correspond à la zone que l'ont souhaite obtenir (dans notre cas : tous les éléments).



Union de 2 ensembles

Exemple

Imaginons une entreprise qui possède plusieurs magasins et dans chacun de ces magasins il y a une table qui liste les clients.

La table du magasin n°1 s'appelle « magasin1_client » et contient les données suivantes :

prenom	nom	ville	date_naissance	total_achat
Léon	Dupuis	Paris	1983-03-06	135
Marie	Bernard	Paris	1993-07-03	75
Sophie	Dupond	Marseille	1986-02-22	27
Marcel	Martin	Paris	1976-11-24	39

La table du magasin n°2 s'appelle « magasin2_client » et contient les données suivantes :

prenom	nom	ville	date_naissance	total_achat
Marion	Leroy	Lyon	1982-10-27	285
Paul	Moreau	Lyon	1976-04-19	133
Marie	Bernard	Paris	1993-07-03	75
Marcel	Martin	Paris	1976-11-24	39

Sachant que certains clients sont présents dans les 2 tables, pour éviter de retourner plusieurs fois les mêmes enregistrement, il convient d'utiliser la requête UNION. La requête SQL est alors la suivante :

```
SELECT * FROM magasin1_client
UNION
SELECT * FROM magasin2_client
```

Résultat :

prenom	nom	ville	date_naissance	total_achat
Léon	Dupuis	Paris	1983-03-06	135
Marie	Bernard	Paris	1993-07-03	75
Sophie	Dupond	Marseille	1986-02-22	27
Marcel	Martin	Paris	1976-11-24	39
Marion	Leroy	Lyon	1982-10-27	285
Paul	Moreau	Lyon	1976-04-19	133

Le résultat de cette requête montre bien que les enregistrements des 2 requêtes sont mis bout-à-bout mais sans inclure plusieurs fois les mêmes lignes.

SQL UNION ALL

La commande UNION ALL de SQL est très similaire à la commande UNION. Elle permet de concaténer les enregistrements de plusieurs requêtes, à la seule différence que cette commande permet d'inclure tous les enregistrements, même les doublons. Ainsi, si un même enregistrement est présents normalement dans les résultats des 2 requêtes concaténées, alors l'union des 2 avec UNION ALL retournera 2 fois ce même résultat.

A savoir : tout comme la commande UNION, il convient que les 2 requêtes retournes exactement le même nombre de colonnes, avec les mêmes types de données et dans le même ordre.

Syntaxe

La syntaxe de la requête SQL pour unir les résultats des 2 tables est la suivante :

```
SELECT * FROM table1
UNION ALL
SELECT * FROM table2
```

Exemple

Imaginons une entreprise qui possède des bases de données dans chacun de ces magasins. Sur ces bases de données il y a une table de la liste des clients avec quelques informations et le total des achats dans l'entreprise.

La table « magasin1_client » correspond au premier magasin :

prenom	nom	ville	date_naissance	total_achat
Léon	Dupuis	Paris	1983-03-06	135
Marie	Bernard	Paris	1993-07-03	75
Sophie	Dupond	Marseille	1986-02-22	27
Marcel	Martin	Paris	1976-11-24	39

La table « magasin2_client » correspond au deuxième magasin :

prenom	nom	ville	date_naissance	total_achat
Marion	Leroy	Lyon	1982-10-27	285
Paul	Moreau	Lyon	1976-04-19	133
Marie	Bernard	Paris	1993-07-03	75
Marcel	Martin	Paris	1976-11-24	39

Pour concaténer les tous les enregistrements de ces tables, il est possible d'effectuer une seule requête utilisant la commande UNION ALL, comme l'exemple ci-dessous :

```
SELECT * FROM magasin1_client
UNION ALL
SELECT * FROM magasin2_client
```

Résultat :

prenom	nom	ville	date_naissance	total_achat
Léon	Dupuis	Paris	1983-03-06	135
Marie	Bernard	Paris	1993-07-03	75
Sophie	Dupond	Marseille	1986-02-22	27
Marcel	Martin	Paris	1976-11-24	39
Marion	Leroy	Lyon	1982-10-27	285
Paul	Moreau	Lyon	1976-04-19	133
Marie	Bernard	Paris	1993-07-03	75
Marcel	Martin	Paris	1976-11-24	39

Le résultat de cette requête montre qu’il y a autant d’enregistrement que dans les 2 tables réunis. A savoir, il y a quelques clients qui étaient présents dans les 2 tables d’origines en conséquent ils sont présent 2 fois dans le résultat de cette requête SQL.

SQL INTERSECT

La commande SQL INTERSECT permet d’obtenir l’intersection des résultats de 2 requêtes. Cette commande permet donc de récupérer les enregistrements communs à 2 requêtes. Cela peut s’avérer utile lorsqu’il faut trouver s’il y a des données similaires sur 2 tables distinctes.

A savoir : pour l’utiliser convenablement il faut que les 2 requêtes retourne le même nombre de colonnes, avec les mêmes types et dans le même ordre.

Syntaxe

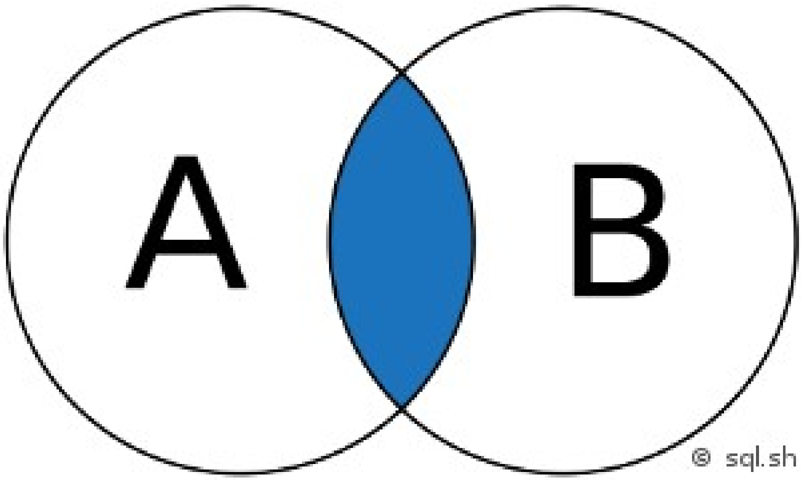
La syntaxe à adopter pour utiliser cette commande est la suivante :

```
SELECT * FROM table1
INTERSECT
SELECT * FROM table2
```

Dans cet exemple, il faut que les 2 tables soient similaires (mêmes colonnes, mêmes types et même ordre). Le résultat correspondra aux enregistrements qui existent dans table1 et dans table2.

Schéma explicatif

L’intersection de 2 ensembles A et B correspond aux éléments qui sont présent dans A et dans B, et seulement ceux-là. Cela peut être représenté par un schéma explicatif simple ou l’intersection de A et B correspond à la zone en bleu.



Intersection de 2 ensembles

Exemple

Prenons l’exemple de 2 magasins qui appartiennent au même groupe. Chaque magasin possède sa table de clients.

La table du magasin n°1 est « magasin1_client » :

prenom	nom	ville	date_naissance	total_achat
Léon	Dupuis	Paris	1983-03-06	135
Marie	Bernard	Paris	1993-07-03	75

Sophie	Dupond	Marseille	1986-02-22	27
Marcel	Martin	Paris	1976-11-24	39

La table du magasin n°2 est « magasin2_client » :

prenom	nom	ville	date_naissance	total_achat
Marion	Leroy	Lyon	1982-10-27	285
Paul	Moreau	Lyon	1976-04-19	133
Marie	Bernard	Paris	1993-07-03	75
Marcel	Martin	Paris	1976-11-24	39

Pour obtenir la liste des clients qui sont présents de façon identiques dans ces 2 tables, il est possible d'utiliser la commande INTERSECT de la façon suivante :

```
SELECT * FROM magasin1_client
INTERSECT
SELECT * FROM magasin2_client
```

Résultat :

prenom	nom	ville	date_naissance	total_achat
Marie	Bernard	Paris	1993-07-03	75
Marcel	Martin	Paris	1976-11-24	39

Le résultat présente 2 enregistrements, il s'agit des clients qui sont à la fois dans la table « magasin1_client » et dans la table « magasin2_client ». Sur certains systèmes une telle requête permet de déceler des erreurs et d'enregistrer seulement à un seul endroit la même information.

SQL EXCEPT / MINUS

Dans le langage SQL la commande EXCEPT s’utilise entre 2 instructions pour récupérer les enregistrements de la première instruction sans inclure les résultats de la seconde requête. Si un même enregistrement devait être présent dans les résultats des 2 syntaxes, ils ne seront pas présent dans le résultat final.

A savoir : cette commande s’appelle différemment selon les Systèmes de Gestion de Base de Données (SGBD) :

- **EXCEPT :** PostgreSQL
- **MINUS :** MySQL et Oracle

Dès lors, il faut remplacer tout le reste de ce cours par MINUS pour les SGBD correspondants.

Syntaxe

La syntaxe d’une requête SQL est toute simple :

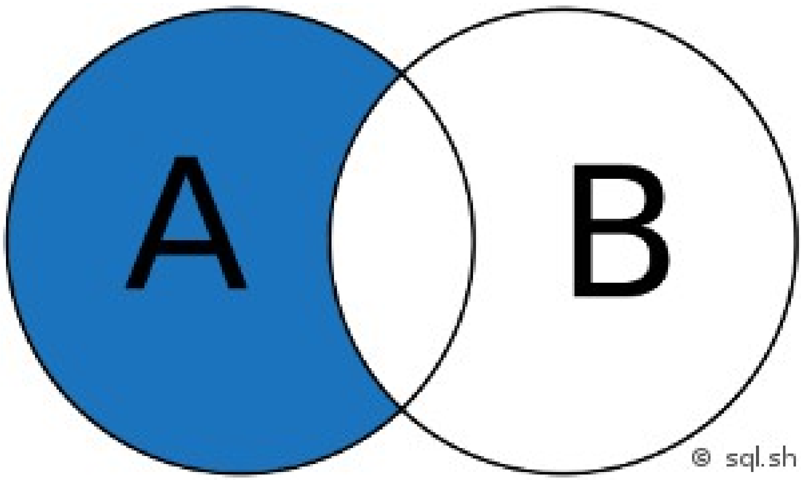
```
SELECT * FROM table1
EXCEPT
SELECT * FROM table2
```

Cette requête permet de lister les résultats du table 1 sans inclure les enregistrements de la table 1 qui sont aussi dans la table 2.

Attention : les colonnes de la première requête doivent être similaires entre la première et la deuxième requête (même nombre, même type et même ordre).

Schéma explicatif

Cette commande permet de récupérer les éléments de l’ensemble A sans prendre en compte les éléments de A qui sont aussi présent dans l’ensemble B. Dans le schéma ci-dessous seule la zone bleu sera retournée grâce à la commande EXCEPT (ou MINUS).



Sélection d’un ensemble avec exception

Exemple

Imaginons un système informatique d’une entreprise. Ce système contient 2 tables contenant des

listes de clients :

- Une table « clients_inscrits » qui contient les prénoms, noms et date d'inscription de clients
- Une table « clients_refus_email » qui contient les informations des clients qui ne souhaitent pas être contacté par email

Cet exemple aura pour objectif de sélectionner les utilisateurs pour envoyer un email d'information. Les utilisateurs de la deuxième table ne devront pas apparaître dans les résultats.

Table « clients_inscrits » :

id	prenom	nom	date_inscription
1	Lionel	Martineau	2012-11-14
2	Paul	Cornu	2012-12-15
3	Sarah	Schmitt	2012-12-17
4	Sabine	Lenoir	2012-12-18

Table « clients_refus_email » :

id	prenom	nom	date_inscription
1	Paul	Cornu	2013-01-27
2	Manuel	Guillot	2013-01-27
3	Sabine	Lenoir	2013-01-29
4	Natalie	Petitjean	2013-02-03

Pour pouvoir sélectionner uniquement le prénom et le nom des utilisateurs qui accepte de recevoir des emails informatifs. La requête SQL à utiliser est la suivante :

```
SELECT prenom, nom FROM clients_inscrits
EXCEPT
SELECT prenom, nom FROM clients_refus_email
```

Résultats :

prenom	nom
Lionel	Martineau
Sarah	Schmitt

Ce tableau de résultats montre bien les utilisateurs qui sont dans inscrits et qui ne sont pas présent dans le deuxième tableau. Par ailleurs, les résultats du deuxième tableau ne sont pas présent sur ce résultat final.

SQL INSERT INTO

L'insertion de données dans une table s'effectue à l'aide de la commande INSERT INTO. Cette commande permet au choix d'inclure une seule ligne à la base existante ou plusieurs lignes d'un coup.

Insertion d'une ligne à la fois

Pour insérer des données dans une base, il y a 2 syntaxes principales :

- Insérer une ligne en indiquant les informations pour chaque colonne existante (en respectant l'ordre)
- Insérer une ligne en spécifiant les colonnes que vous souhaiter compléter. Il est possible d'insérer une ligne en renseigner seulement une partie des colonnes

Insérer une ligne en spécifiant toutes les colonnes

La syntaxe pour remplir une ligne avec cette méthode est la suivante :

```
INSERT INTO table
VALUES ('valeur 1', 'valeur 2', ...)
```

Cette syntaxe possède les avantages et inconvénients suivants :

- Obliger de remplir toutes les données, tout en respectant l'ordre des colonnes
- Il n'y a pas le nom de colonne, donc les fautes de frappe sont limitées. Par ailleurs, les colonnes peuvent être renommées sans avoir à changer la requête
- L'ordre des colonnes doit resté identique sinon certaines valeurs prennent le risque d'être complétée dans la mauvaise colonne

Insérer une ligne en spécifiant seulement les colonnes souhaitées

Cette deuxième solution est très similaire, excepté qu'il faut indiquer le nom des colonnes avant « VALUES ». La syntaxe est la suivante :

```
INSERT INTO table
(nom_colonne_1, nom_colonne_2, ...
VALUES ('valeur 1', 'valeur 2', ...)
```

A noter : il est possible de ne pas renseigner toutes les colonnes. De plus, l'ordre des colonnes n'est pas important.

Insertion de plusieurs lignes à la fois

Il est possible d'ajouter plusieurs lignes à un tableau avec une seule requête. Pour ce faire, il convient d'utiliser la syntaxe suivante :

```
INSERT INTO client (prenom, nom, ville, age)
 VALUES
 ('Rébecca', 'Armand', 'Saint-Didier-des-Bois', 24),
 ('Aimée', 'Hebert', 'Marigny-le-Châtel', 36),
 ('Marielle', 'Ribeiro', 'Maillères', 27),
```

```
('Hilaire', 'Savary', 'Conie-Molitard', 58);
```

A noter : lorsque le champ à remplir est de type VARCHAR ou TEXT il faut indiquer le texte entre guillemet simple. En revanche, lorsque la colonne est un numérique tel que INT ou BIGINT il n’y a pas besoin d’utiliser de guillemet, il suffit juste d’indiquer le nombre.

Un tel exemple sur une table vide va créer le tableau suivant :

id	prenom	nom	ville	age
1	Rébecca	Armand	Saint-Didier-des-Bois	24
2	Aimée	Hebert	Marigny-le-Châtel	36
3	Marielle	Ribeiro	Maillères	27
4	Hilaire	Savary	Conie-Molitard	58

SQL ON DUPLICATE KEY UPDATE

L'instruction ON DUPLICATE KEY UPDATE est une fonctionnalité de MySQL qui permet de mettre à jour des données lorsqu'un enregistrement existe déjà dans une table. Cela permet d'avoir qu'une seule requête SQL pour effectuer selon la convenance un INSERT ou un UPDATE.

Syntaxe

Cette commande s'effectue au sein de la requête INSERT INTO avec la syntaxe suivante :

```
INSERT INTO table (a, b, c)
VALUES (1, 20, 68)
ON DUPLICATE KEY UPDATE a=a+1
```

A noter : cette requête se traduit comme suit :

1. insérer les données a, b et c avec les données respectives de 1, 20 et 68
2. Si la clé primaire existe déjà pour ces valeurs alors seulement faire une mise à jour de a = a+1

Exemple avec la commande WHERE

Grâce à la commande « ON DUPLICATE KEY » Il est possible d'enregistrer la date à laquelle la données est insérée pour la première fois et la date de dernière mise à jour, comme le montre la commande ci-dessous :

```
INSERT INTO table (a, b, c, date_insert)
VALUES (1, 20, 1, NOW())
ON DUPLICATE KEY UPDATE date_update=NOW
WHERE c=1
```

A noter : cette requête se traduit comme suit :

1. insérer les données a, b, c et date_insert, avec les données respectives de 1, 20, 1 ainsi que la date et l'heure actuelle
2. Si la clé primaire existe déjà pour ces valeurs alors mettre a jour la date et l'heure du champ « date_update »
3. Effectuer la mise à jour uniquement sur les champs où c = 1

Exemple

Imaginons une application qui laisse les utilisateurs voter pour les produits qu'ils préfèrent. Le système de vote est très simple et est basé sur des +1. La table des votes contient le nombre de votes par produits avec la date du premier vote et la date du dernier vote.

Table vote :

id	produit_id	vote_count	vote_first_date	vote_last_date
1	46	2	2012-04-25 17:45:24	2013-02-16 09:47:02
2	39	4	2012-04-28 16:54:44	2013-02-14 21:04:35
3	49	1	2012-04-25 19:11:09	2013-01-06 20:32:57

Pour n’utiliser qu’une seule ligne qui permet d’ajouter des votes dans cette table, sans se préoccuper de savoir s’il faut faire un INSERT ou un UPDATE, il est possible d’utiliser la requête SQL suivante :

```
INSERT INTO vote (produit_id, vote_count, vote_first_date, vote_last_date)
VALUES (50, 1, NOW(), NOW())
ON DUPLICATE KEY UPDATE vote_count = vote_count+1, vote_last_date = NOW()
```

Dans cette requête la date et l’heure est générée automatiquement avec la fonction NOW().

Résultat après la première exécution de la requête :

id	produit_id	vote_count	vote_first_date	vote_last_date
1	46	2	2012-04-25 17:45:24	2013-02-16 09:47:02
2	39	4	2012-04-28 16:54:44	2013-02-14 21:04:35
3	49	1	2012-04-25 19:11:09	2013-01-06 20:32:57
4	55	1	2013-04-02 15:06:34	2013-04-02 15:06:34

Ce résultat montre bien l’ajout d’une ligne en fin de table, donc la requête a été utilisé sous la forme d’un INSERT. Après une deuxième exécution de cette même requête le lendemain, les données seront celles-ci:

id	produit_id	vote_count	vote_first_date	vote_last_date
1	46	2	2012-04-25 17:45:24	2013-02-16 09:47:02
2	39	4	2012-04-28 16:54:44	2013-02-14 21:04:35
3	49	1	2012-04-25 19:11:09	2013-01-06 20:32:57
4	55	2	2013-04-02 15:06:34	2013-04-03 08:14:57

Ces résultats montre bien qu’il y a eu un vote supplémentaire et que la date du dernier vote a été mis à jour.

Insérer une ligne ou ne rien faire

Dans certains cas il est intéressant d’utiliser un INSERT mais de ne rien faire si la commande a déjà été insérée précédemment. Malheureusement, si la clé primaire existe déjà la requête retournera une erreur. Et s’il n’y a rien à mettre à jour, la commande ON DUPLICATE KEY UPDATE (ODKU) ne semble pas convenir. Toutefois il y a une astuce qui consiste à utiliser une requête de ce type :

```
INSERT INTO table (a, b, c)
VALUES (1, 45, 6)
ON DUPLICATE KEY UPDATE id = id
```

Cette requête insert les données et ne produit aucune erreur si l’enregistrement existait déjà dans la table.

A savoir : théoriquement il aurait été possible d’utiliser INSERT IGNORE mais malheureusement cela

empêche de retourner des erreurs telles que des erreurs de conversions de données.

Compatibilité

Pour le moment cette fonctionnalité n’est possible qu’avec MySQL depuis la version 4.1 (date de 2003). Les autres Systèmes de Gestion de Bases de Données (SGBD) n’intègrent pas cette fonctionnalité. Pour simuler cette fonctionnalité il y a quelques alternatives :

- **PostgreSQL :** il y a une astuce en utilisant une fonction. L’astuce est expliquée dans la documentation officielle : fonction INSERT/UPDATE.
- **Oracle :** il est possible d’utiliser la commande MERGE pour effectuer la même chose.
- **SQL Server :** il est possible d’utiliser une procédure.

SQL UPDATE

La commande UPDATE permet d'effectuer des modifications sur des lignes existantes. Très souvent cette commande est utilisée avec WHERE pour spécifier sur quelles lignes doivent porter la ou les modifications.

Syntaxe

La syntaxe basique d'une requête utilisant UPDATE est la suivante :

```
UPDATE table
SET nom_colonne_1 = 'nouvelle valeur'
WHERE condition
```

Cette syntaxe permet d'attribuer une nouvelle valeur à la colonne nom_colonne_1 pour les lignes qui respectent la condition stipulé avec WHERE. Il est aussi possible d'attribuer la même valeur à la colonne nom_colonne_1 pour toutes les lignes d'une table si la condition WHERE n'était pas utilisée.

A noter, pour spécifier en une seule fois plusieurs modification, il faut séparer les attributions de valeur par des virgules. Ainsi la syntaxe deviendrait la suivante :

```
UPDATE table
SET colonne_1 = 'valeur 1', colonne_2 = 'valeur 2', colonne_3 = 'valeur 3'
WHERE condition
```

Exemple

id	nom	rue	ville	code_postal	pays
1	Chantal	12 Avenue du Petit Trianon	Puteaux	92800	France
2	Pierre	18 Rue de l'Allier	Ponthion	51300	France
3	Romain	3 Chemin du Chiron	Trévérien	35190	France

Imaginons une table « client » qui présente les coordonnées de clients.

Pour modifier l'adresse du client Pierre, il est possible d'utiliser la requête suivante :

```
UPDATE client
SET rue = '49 Rue Ameline',
  ville = 'Saint-Eustache-la-Forêt',
  code_postal = '76210'
WHERE id = 2
```

Résultat :

id	nom	rue	ville	code_postal	pays
1	Chantal	12 Avenue du Petit Trianon	Puteaux	92800	France
2	Pierre	49 Rue Ameline	Saint-Eustache-la-Forêt	76210	France
3	Romain	3 Chemin du Chiron	Trévérien	35190	France

SQL DELETE

La commande DELETE en SQL permet de supprimer des lignes dans une table. En utilisant cette commande associé à WHERE il est possible de sélectionner les lignes concernées qui seront supprimées.

Attention : Avant d'essayer de supprimer des lignes, il est recommandé d'effectuer une sauvegarde de la base de données, ou tout du moins de la table concernée par la suppression. Ainsi, s'il y a une mauvaise manipulation il est toujours possible de restaurer les données.

Syntaxe

La syntaxe pour supprimer des lignes est la suivante :

```
DELETE FROM table
WHERE condition
```

Attention : s'il n'y a pas de condition WHERE alors toutes les lignes seront supprimées et la table sera alors vide.

Exemple

Imaginons une table « utilisateur » qui contient des informations sur les utilisateurs d'une application.

id	nom	prenom	date_inscription
1	Bazin	Daniel	2012-02-13
2	Favre	Constantin	2012-04-03
3	Clerc	Guillaume	2012-04-12
4	Ricard	Rosemonde	2012-06-24
5	Martin	Natalie	2012-07-02

Si l'ont souhaite supprimer les utilisateurs qui se sont inscrit avant le « 10/04/2012″, il va falloir effectuer la requête suivante :

```
DELETE FROM utilisateur
WHERE date_inscription < '2012-04-10'
```

La requête permettra alors de supprimer les utilisateurs « Daniel » et « Constantin ». La table contiendra alors les données suivantes :

id	nom	prenom	date_inscription
3	Clerc	Guillaume	2012-04-12
4	Ricard	Rosemonde	2012-06-24
5	Martin	Natalie	2012-07-02

Il ne faut pas oublier qu'il est possible d'utiliser d'autres conditions pour sélectionner les lignes à supprimer.

SQL MERGE

Dans le langage SQL, la commande MERGE permet d'insérer ou de mettre à jour des données dans une table. Cette commande permet d'éviter d'effectuer plusieurs requêtes pour savoir si une donnée est déjà dans la base de données et ainsi adapter l'utilisation d'une requête pour ajouter ou une autre pour modifier la donnée existante. Cette commande peut aussi s'appeler « upsert ».

Attention : bien que l'instruction a été ajoutée dans le standard SQL:2003, les différentes SGBD n'utilisent pas toutes les mêmes méthodes pour effectuer un upsert.

Syntaxe

La syntaxe standard pour effectuer un merge consiste à utiliser une requête SQL semblable à celle ci-dessous :

```
MERGE INTO table1
  USING table_reference
  ON (conditions)
  WHEN MATCHED THEN
    UPDATE SET table1.colonne1 = valeur1, table1.colonne2 = valeur2
    DELETE WHERE conditions2
  WHEN NOT MATCHED THEN
    INSERT (colonnes1, colonne3)
    VALUES (valeur1, valeur3)
```

Voici les explications détaillées de cette requête :

- MERGE INTO permet de sélectionner la table à modifier
- USING et ON permet de lister les données sources et la condition de correspondance
- WHEN MATCHED permet de définir la condition de mise à jour lorsque la condition est vérifiée
- WHEN NOT MATCHED permet de définir la condition d'insertion lorsque la condition n'est pas vérifiée

Compatibilité

Les systèmes de gestion de bases de données peuvent implémenter cette fonctionnalité soit de façon standard, en utilisant une commande synonyme ou en utilisant une syntaxe non standard.

- **Syntaxe standard :** SQL Server, Oracle, DB2, Teradata et EXASOL
- **Utilisation du terme UPSERT :** Microsoft SQL Azure et MongoDB
- **Utilisation non standard :** MySQL, SQLite, Firebird, IBM DB2 et Microsoft SQL

SQL TRUNCATE TABLE

En SQL, la commande TRUNCATE permet de supprimer toutes les données d’une table sans supprimer la table en elle-même. En d’autres mots, cela permet de purger la table. Cette instruction diffère de la commande DROP qui à pour but de supprimer les données ainsi que la table qui les contient.

A noter : l’instruction TRUNCATE est semblable à l’instruction DELETE sans utilisation de WHERE. Parmi les petite différences TRUNCATE est toutefois plus rapide et utilise moins de ressource. Ces gains en performance se justifie notamment parce que la requête n’indiquera pas le nombre d’enregistrement supprimés et qu’il n’y aura pas d’enregistrement des modifications dans le journal.

Syntaxe

Cette instruction s’utilise dans une requête SQL semblable à celle-ci :

```
TRUNCATE TABLE `table`
```

Dans cet exemple, les données de la table « table » seront perdues une fois cette requête exécutée.

Exemple

Pour montrer un exemple concret de l’utilisation de cette commande, nous pouvons imaginez un système informatique contenant la liste des fournitures d’une entreprise. Ces données seraient tout simplement stockées dans une table « fourniture ».

Table « fourniture » :

id	nom	date_ajout
1	Ordinateur	2013-04-05
2	Chaise	2013-04-14
3	Bureau	2013-07-18
4	Lampe	2013-09-27

Il est possible de supprimer toutes les données de cette table en utilisant la requête suivante :

```
TRUNCATE TABLE `fourniture`
```

Une fois la requête exécutée, la table ne contiendra plus aucun enregistrement. En d’autres mots, toutes les lignes du tableau présenté ci-dessus auront été supprimées.

SQL CREATE DATABASE

La création d'une base de données en SQL est possible en ligne de commande. Même si les systèmes de gestion de base de données (SGBD) sont souvent utilisés pour créer une base, il convient de connaître la commande à utiliser, qui est très simple.

Syntaxe

Pour créer une base de données qui sera appelé « ma_base » il suffit d'utiliser la requête suivante qui est très simple :

```
CREATE DATABASE ma_base
```

Base du même nom qui existe déjà

Avec MySQL, si une base de données porte déjà ce nom, la requête retournera une erreur. Pour éviter d'avoir cette erreur, il convient d'utiliser la requête suivante pour MySQL :

```
CREATE DATABASE IF NOT EXISTS ma_base
```

L'option IF NOT EXISTS permet juste de ne pas retourner d'erreur si une base du même nom existe déjà. La base de données ne sera pas écrasée.

Options

Dans le standard SQL la commande CREATE DATABASE n'existe normalement pas. En conséquent il revient de vérifier la documentation des différents SGBD pour vérifier les syntaxes possibles pour définir des options. Ces options permettent selon les cas, de définir les jeux de caractères, le propriétaire de la base ou même les limites de connexion.

SQL DROP DATABASE

En SQL, la commande DROP DATABASE permet de supprimer totalement une base de données et tout ce qu'elle contient. Cette commande est à utiliser avec beaucoup d'attention car elle permet de supprimer tout ce qui est inclus dans une base: les tables, les données, les index …

Syntaxe

Pour supprimer la base de données « ma_base », la requête est la suivante :

```
DROP DATABASE ma_base
```

Attention : cela va supprimer toutes les tables et toutes les données de cette base. Si vous n'êtes pas sûr de ce que vous faites, n'hésitez pas à effectuer une sauvegarde de la base avant de supprimer.

Ne pas afficher d'erreur si la base n'existe pas

Par défaut, si le nom de base utilisé n'existe pas, la requête retournera une erreur. Pour éviter d'obtenir cette erreur si vous n'êtes pas sûr du nom, il est possible d'utiliser l'option IF EXISTS. La syntaxe sera alors la suivante :

```
DROP DATABASE IF EXISTS ma_base
```

SQL CREATE TABLE

La commande CREATE TABLE permet de créer une table en SQL. Un tableau est une entité qui est contenu dans une base de données pour stocker des données ordonnées dans des colonnes. La création d'une table sert à définir les colonnes et le type de données qui seront contenus dans chacun des colonne (entier, chaîne de caractères, date, valeur binaire …).

Syntaxe

La syntaxe générale pour créer une table est la suivante :

```
CREATE TABLE nom_de_la_table
(
    colonne1 type_donnees,
    colonne2 type_donnees,
    colonne3 type_donnees,
    colonne4 type_donnees
)
```

Dans cette requête, 4 colonnes ont été définies. Le mot-clé « type_donnees » sera à remplacer par un mot-clé pour définir le type de données (INT, DATE, TEXT …). Pour chaque colonne, il est également possible de définir des options telles que (liste non-exhaustive) :

- **NOT NULL :** empêche d'enregistrer une valeur nulle pour une colonne.
- **DEFAULT :** attribuer une valeur par défaut si aucune données n'est indiquée pour cette colonne lors de l'ajout d'une ligne dans la table.
- **PRIMARY KEY :** indiquer si cette colonne est considérée comme clé primaire pour un index.

Exemple

Imaginons que l'ont souhaite créer une table utilisateur, dans laquelle chaque ligne correspond à un utilisateur inscrit sur un site web. La requête pour créer cette table peut ressembler à ceci :

```
CREATE TABLE utilisateur
(
    id INT PRIMARY KEY NOT NULL,
    nom VARCHAR(100),
    prenom VARCHAR(100),
    email VARCHAR(255),
    date_naissance DATE,
    pays VARCHAR(255),
    ville VARCHAR(255),
    code_postal VARCHAR(5),
    nombre_achat INT
)
```

Voici des explications sur les colonnes créées :

- **id :** identifiant unique qui est utilisé comme clé primaire et qui n'est pas nulle
- **nom :** nom de l'utilisateur dans une colonne de type VARCHAR avec un maximum de 100

caractères au maximum

- **prenom :** idem mais pour le prénom
- **email :** adresse email enregistré sous 255 caractères au maximum
- **date_naissance :** date de naissance enregistré au format AAAA-MM-JJ (exemple : 1973-11-17)
- **pays :** nom du pays de l’utilisateur sous 255 caractères au maximum
- **ville :** idem pour la ville
- **code_postal :** 5 caractères du code postal
- **nombre_achat :** nombre d’achat de cet utilisateur sur le site

SQL ALTER TABLE

La commande ALTER TABLE en SQL permet de modifier une table existante. Il est ainsi possible d'ajouter une colonne, d'en supprimer une ou de modifier une colonne existante, par exemple pour changer le type.

Syntaxe de base

D'une manière générale, la commande s'utilise de la manière suivante :

```
ALTER TABLE nom_table
instruction
```

Le mot-clé « instruction » ici sert à désigner une commande supplémentaire, qui sera détaillée ci-dessous selon l'action que l'ont souhaite effectuer : ajouter, supprimer ou modifier une colonne.

Ajouter une colonne

Syntaxe

L'ajout d'une colonne dans une table est relativement simple et peut s'effectuer à l'aide d'une requête ressemblant à ceci :

```
ALTER TABLE nom_table
ADD nom_colonne type_donnees
```

Exemple

Pour ajouter une colonne qui correspond à une rue sur une table utilisateur, il est possible d'utiliser la requête suivante :

```
ALTER TABLE utilisateur
ADD adresse_rue VARCHAR(255)
```

Supprimer une colonne

Une syntaxe permet également de supprimer une colonne pour une table. Il y a 2 manières totalement équivalente pour supprimer une colonne :

```
ALTER TABLE nom_table
DROP nom_colonne
```

Ou (le résultat sera le même)

```
ALTER TABLE nom_table
DROP COLUMN nom_colonne
```

Modifier une colonne

Pour modifier une colonne, comme par exemple changer le type d'une colonne, il y a différentes syntaxes selon le SGBD.

MySQL

```
ALTER TABLE nom_table
MODIFY nom_colonne type_donnees
```

PostgreSQL

```
ALTER TABLE nom_table
ALTER COLUMN nom_colonne TYPE type_donnees
```

Ici, le mot-clé « type_donnees » est à remplacer par un type de données tel que INT, VARCHAR, TEXT, DATE …

Renommer une colonne

Pour renommer une colonne, il convient d'indiquer l'ancien nom de la colonne et le nouveau nom de celle-ci.

MySQL

Pour MySQL, il faut également indiquer le type de la colonne.

```
ALTER TABLE nom_table
CHANGE colonne_ancien_nom colonne_nouveau_nom type_donnees
```

Ici « type_donnees » peut correspondre par exemple à INT, VARCHAR, TEXT, DATE …

PostgreSQL

Pour PostgreSQL la syntaxe est plus simple et ressemble à ceci (le type n'est pas demandé) :

```
ALTER TABLE nom_table
RENAME COLUMN colonne_ancien_nom TO colonne_nouveau_nom
```

SQL DROP TABLE

La commande DROP TABLE en SQL permet de supprimer définitivement une table d'une base de données. Cela supprime en même temps les éventuels index, trigger, contraintes et permissions associées à cette table.

Attention : il faut utiliser cette commande avec attention car une fois supprimée, les données sont perdues. Avant de l'utiliser sur une base importante il peut être judicieux d'effectuer un backup (une sauvegarde) pour éviter les mauvaises surprises.

Syntaxe

Pour supprimer une table « nom_table » il suffit simplement d'utiliser la syntaxe suivante :

```
DROP TABLE nom_table
```

A savoir : s'il y a une dépence avec une autre table, il est recommandé de les supprimer avant de supprimer la table. C'est le cas par exemple s'il y a des clés étrangères.

Intérêts

Il arrive qu'une table soit créé temporairement pour stoquer des données qui n'ont pas vocation à être ré-utiliser. La suppression d'une table non utilisée est avantageux sur plusieurs aspects :

- Libérer de la mémoire et alléger le poids des backups
- Éviter des erreurs dans le futur si une table porte un nom similaire ou qui porte à confusion
- Lorsqu'un développeur ou administrateur de base de données découvre une application, il est plus rapide de comprendre le système s'il n'y a que les tables utilisées qui sont présente

Exemple de requête

Imaginons qu'une base de données possède une table « client_2009 » qui ne sera plus jamais utilisé et qui existe déjà dans un ancien backup. Pour supprimer cette table, il suffit d'effectuer la requête suivante :

```
DROP TABLE client_2009
```

L'exécution de cette requête va permettre de supprimer la table.

Jointure SQL

Les jointures en SQL permettent d'associer plusieurs tables dans une même requête. Cela permet d'exploiter la puissance des bases de données relationnelles pour obtenir des résultats qui combinent les données de plusieurs tables de manière efficace.

Exemple

En général, les jointures consistent à associer des lignes de 2 tables en associant l'égalité des valeurs d'une colonne d'une première table par rapport à la valeur d'une colonne d'une seconde table. Imaginons qu'une base de 2 données possède une table « utilisateur » et une autre table « adresse » qui contient les adresses de ces utilisateurs. Avec une jointure, il est possible d'obtenir les données de l'utilisateur et de son adresse en une seule requête.

On peut aussi imaginer qu'un site web possède une table pour les articles (titre, contenu, date de publication …) et une autre pour les rédacteurs (nom, date d'inscription, date de naissance …). Avec une jointure il est possible d'effectuer une seule recherche pour afficher un article et le nom du rédacteur. Cela évite d'avoir à afficher le nom du rédacteur dans la table « article ».

Il y a d'autres cas de jointures, incluant des jointures sur la même table ou des jointure d'inégalité. Ces cas étant assez particulier et pas si simple à comprendre, ils ne seront pas élaboré sur cette page.

Types de jointures

Il y a plusieurs méthodes pour associer 2 tables ensemble. Voici la liste des différentes techniques qui sont utilisées :

- **INNER JOIN :** jointure interne pour retourner les enregistrements quand la condition est vrai dans les 2 tables. C'est l'une des jointures les plus communes.
- **CROSS JOIN :** jointure croisée permettant de faire le produit cartésien de 2 tables. En d'autres mots, permet de joindre chaque lignes d'une table avec chaque lignes d'une seconde table. Attention, le nombre de résultats est en général très élevé.
- **LEFT JOIN (ou LEFT OUTER JOIN) :** jointure externe pour retourner tous les enregistrements de la table de gauche (LEFT = gauche) même si la condition n'est pas vérifié dans l'autre table.
- **RIGHT JOIN (ou RIGHT OUTER JOIN) :** jointure externe pour retourner tous les enregistrements de la table de droite (RIGHT = droite) même si la condition n'est pas vérifié dans l'autre table.
- **FULL JOIN (ou FULL OUTER JOIN) :** jointure externe pour retourner les résultats quand la condition est vrai dans au moins une des 2 tables.
- **SELF JOIN :** permet d'effectuer une jointure d'une table avec elle-même comme si c'était une autre table.
- **NATURAL JOIN :** jointure naturelle entre 2 tables s'il y a au moins une colonne qui porte le même nom entre les 2 tables SQL
- **UNION JOIN :** jointure d'union

SQL INNER JOIN

Dans le langage SQL la commande INNER JOIN, aussi appelée EQUIJOIN, est un type de jointures très communes pour lier plusieurs tables entre-elles. Cette commande retourne les enregistrements lorsqu'il y a au moins une ligne dans chaque colonne qui correspond à la condition.

Syntaxe

Pour utiliser ce type de jointure il convient d'utiliser une requête SQL avec cette syntaxe :

```
SELECT *
FROM table1
INNER JOIN table2 ON table1.id = table2.fk_id
```

La syntaxe ci-dessus stipule qu'il faut sélectionner les enregistrements des tables table1 et table2 lorsque les données de la colonne « id » de table1 est égal aux données de la colonne fk_id de table2.

La jointure SQL peux aussi être écrite de la façon suivante :

```
SELECT *
FROM table1
INNER JOIN table2
WHERE table1.id = table2.fk_id
```

La syntaxe avec la condition WHERE est une manière alternative de faire la jointure mais qui possède l'inconvénient d'être moins facile à lire s'il y a déjà plusieurs conditions dans le WHERE.

Exemple

Imaginons une application qui possède une table utilisateur ainsi qu'une table commande qui contient toutes les commandes effectuées par les utilisateurs.

Table utilisateur :

id	prenom	nom	email	ville
1	Aimée	Marechal	aime.marechal@example.com	Paris
2	Esmée	Lefort	esmee.lefort@example.com	Lyon
3	Marine	Prevost	m.prevost@example.com	Lille
4	Luc	Rolland	lucrolland@example.com	Marseille

Table commande :

utilisateur_id	date_achat	num_facture	prix_total
1	2013-01-23	A00103	203.14
1	2013-02-14	A00104	124.00
2	2013-02-17	A00105	149.45
2	2013-02-21	A00106	235.35
5	2013-03-02	A00107	47.58

Pour afficher toutes les commandes associées aux utilisateurs, il est possible d'utiliser la requête suivante :

```
SELECT id, prenom, nom, date_achat, num_facture, prix_total
FROM utilisateur
INNER JOIN commande ON utilisateur.id = commande.utilisateur_id
```

Résultats :

id	prenom	nom	date_achat	num_facture	prix_total
1	Aimée	Marechal	2013-01-23	A00103	203.14
1	Aimée	Marechal	2013-02-14	A00104	124.00
2	Esmée	Lefort	2013-02-17	A00105	149.45
2	Esmée	Lefort	2013-02-21	A00106	235.35

Le résultat de la requête montre parfaite la jointure entre les 2 tables. Les utilisateurs 3 et 4 ne sont pas affichés puisqu'il n'y a pas de commandes associés à ces utilisateurs.

Attention : il est important de noter que si un utilisateur à été supprimé, alors on ne verra pas ses commandes dans la liste puisque INNER JOIN retourne uniquement les résultats ou la condition est vrai dans les 2 tables.

SQL CROSS JOIN

Dans le langage SQL, la commande CROSS JOIN est un type de jointure sur 2 tables SQL qui permet de retourner le produit cartésien. Autrement dit, cela permet de retourner chaque ligne d'une table avec chaque ligne d'une autre table. Ainsi effectuer le produit cartésien d'une table A qui contient 30 résultats avec une table B de 40 résultats va produire 1200 résultats (30 x 40 = 1200). En général la commande CROSS JOIN est combinée avec la commande WHERE pour filtrer les résultats qui respectent certaines conditions.

Attention, le nombre de résultat peut facilement être très élevé. S'il est effectué sur des tables avec beaucoup d'enregistrements, cela peut ralentir sensiblement le serveur.

Syntaxe

Pour effectuer un jointure avec CROSS JOIN, il convient d'effectuer une requête SQL respectant la syntaxe suivante :

```
SELECT *
FROM table1
CROSS JOIN table2
```

Méthode alternative pour retourner les mêmes résultats :

```
SELECT *
FROM table1, table2
```

L'une ou l'autre de ces syntaxes permettent d'associer tous les résultats de table1 avec chacun des résultats de table2.

Exemple

Imaginons une application de recettes de cuisines qui contient 2 tables d'ingrédients, la table legume et la table fruit.

Table legume :

l_id	l_nom_fr_fr	l_nom_en_gb
45	Carotte	Carott
46	Oignon	Onion
47	Poireau	Leek

Table fruit :

f_id	f_nom_fr_fr	f_nom_en_gb
87	Banane	Banana
88	Kiwi	Kiwi
89	Poire	Pear

Pour une raison quelconque l'application doit associer tous les légumes avec tous les fruits. Toutes les combinaisons doivent être affichées. Pour cela il convient d'effectuer l'une ou l'autre des requêtes suivantes :

```
SELECT l_id, l_nom_fr_fr, f_id, f_nom_fr_fr
FROM legume
CROSS JOIN fruit
```

ou :

```
SELECT l_id, l_nom_fr_fr, f_id, f_nom_fr_fr
FROM legume, fruit
```

Résultats :

l_id	l_nom_fr_fr	f_id	f_nom_fr_fr
45	Carotte	87	Banane
45	Carotte	88	Kiwi
45	Carotte	89	Poire
46	Oignon	87	Banane
46	Oignon	88	Kiwi
46	Oignon	89	Poire
47	Poireau	87	Banane
47	Poireau	88	Kiwi
47	Poireau	89	Poire

Le résultat montre bien que chaque légume est associé à chaque fruit. Avec 3 fruits et 3 légumes, il y a donc 9 lignes de résultats (3 x3 = 9).

SQL LEFT JOIN

Dans le langage SQL, la commande LEFT JOIN (aussi appelée LEFT OUTER JOIN) est un type de jointure entre 2 tables. Cela permet de lister tous les résultats de la table de gauche (left = gauche) même s'il n'y a pas de correspondance dans la deuxième tables.

Syntaxe

Pour lister les enregistrement de table1, même s'il n'y a pas de correspondance avec table2, il convient d'effectuer une requête SQL utilisant la syntaxe suivante.

```
SELECT *
FROM table1
LEFT JOIN table2 ON table1.id = table2.fk_id
```

La requête peux aussi s'écrire de la façon suivante :

```
SELECT *
FROM table1
LEFT OUTER JOIN table2 ON table1.id = table2.fk_id
```

Cette requête est particulièrement intéressante pour récupérer les informations de table1 tout en récupérant les données associées, même s'il n'y a pas de correspondance avec table2. A savoir, s'il n'y a pas de correspondance les colonnes de table2 vaudront toutes NULL.

Exemple

Imaginons une application contenant des utilisateurs et des commandes pour chacun de ces utilisateurs. La base de données de cette application contient une table pour les utilisateurs et sauvegarde leurs achats dans une seconde table. Les 2 tables sont reliées grâce à la colonne utilisateur_id de la table des commandes. Cela permet d'associer une commande à un utilisateur.

Table utilisateur :

id	prenom	nom	email	ville
1	Aimée	Marechal	aime.marechal@example.com	Paris
2	Esmée	Lefort	esmee.lefort@example.com	Lyon
3	Marine	Prevost	m.prevost@example.com	Lille
4	Luc	Rolland	lucrolland@example.com	Marseille

Table commande :

utilisateur_id	date_achat	num_facture	prix_total
1	2013-01-23	A00103	203.14
1	2013-02-14	A00104	124.00
2	2013-02-17	A00105	149.45
2	2013-02-21	A00106	235.35
5	2013-03-02	A00107	47.58

Pour lister tous les utilisateurs avec leurs commandes et afficher également les utilisateurs qui n’ont pas effectuées d’achats, il est possible d’utiliser la requête suivante :

```
SELECT *
FROM utilisateur
LEFT JOIN commande ON utilisateur.id = commande.utilisateur_id
```

Résultats :

id	prenom	nom	date_achat	num_facture	prix_total
1	Aimée	Marechal	2013-01-23	A00103	203.14
1	Aimée	Marechal	2013-02-14	A00104	124.00
2	Esmée	Lefort	2013-02-17	A00105	149.45
2	Esmée	Lefort	2013-02-21	A00106	235.35
3	Marine	Prevost	NULL	NULL	NULL
4	Luc	Rolland	NULL	NULL	NULL

Les dernières lignes montrent des utilisateurs qui n’ont effectuée aucune commande. La ligne retourne la valeur NULL pour les colonnes concernant les achats qu’ils n’ont pas effectués.

Filtrer sur la valeur NULL

Attention, la valeur NULL n’est pas une chaîne de caractère. Pour filtrer sur ces caractères il faut utiliser la commande IS NULL. Par exemple, pour lister les utilisateurs qui n’ont pas effectués d’achats il est possible d’utiliser la requête suivante.

```
SELECT id, prenom, nom, utilisateur_id
FROM utilisateur
LEFT JOIN commande ON utilisateur.id = commande.utilisateur_id
WHERE utilisateur_id IS NULL
```

Résultats :

id	prenom	nom	utilisateur_id
3	Marine	Prevost	NULL
4	Luc	Rolland	NULL

SQL RIGHT JOIN

En SQL, la commande RIGHT JOIN (ou RIGHT OUTER JOIN) est un type de jointure entre 2 tables qui permet de retourner tous les enregistrements de la table de droite (right = droite) même s'il n'y a pas de correspondance avec la table de gauche. S'il y a un enregistrement de la table de droite qui ne trouve pas de correspondance dans la table de gauche, alors les colonnes de la table de gauche auront NULL pour valeur.

Syntaxe

L'utilisation de cette commande SQL s'effectue de la façon suivante :

```
SELECT *
FROM table1
RIGHT JOIN table2 ON table1.id = table2.fk_id
```

La syntaxe de cette requête SQL peux aussi s'écrire de la façon suivante :

```
SELECT *
FROM table1
RIGHT OUTER JOIN table2 ON table1.id = table2.fk_id
```

Cette syntaxe stipule qu'il faut lister toutes les lignes du tableau table2 (tableau de droite) et afficher les données associées du tableau table1 s'il y a une correspondance entre ID de table1 et FK_ID de table2. S'il n'y a pas de correspondance, l'enregistrement de table2 sera affiché et les colonnes de table1 vaudront toutes NULL.

Exemple

Prenons l'exemple d'une base de données qui contient des utilisateurs et un historique d'achat de ces utilisateurs. Cette 2 tables sont reliées entre grâce à la colonne utilisateur_id de la table des commandes. Cela permet de savoir à quel utilisateur est associé un achat.

Table utilisateur :

id	prenom	nom	email	ville	actif
1	Aimée	Marechal	aime.marechal@example.com	Paris	1
2	Esmée	Lefort	esmee.lefort@example.com	Lyon	0
3	Marine	Prevost	m.prevost@example.com	Lille	1
4	Luc	Rolland	lucrolland@example.com	Marseille	1

Table commande :

utilisateur_id	date_achat	num_facture	prix_total
1	2013-01-23	A00103	203.14
1	2013-02-14	A00104	124.00
2	2013-02-17	A00105	149.45
3	2013-02-21	A00106	235.35
5	2013-03-02	A00107	47.58

Pour afficher toutes les commandes avec le nom de l'utilisateur correspondant il est normalement d'habitude d'utiliser INNER JOIN en SQL. Malheureusement, si l'utilisateur a été supprimé de la table, alors ça ne retourne pas l'achat. L'utilisation de RIGHT JOIN permet de retourner tous les achats et d'afficher le nom de l'utilisateur s'il existe. Pour cela il convient d'utiliser cette requête :

```
SELECT id, prenom, nom, utilisateur_id, date_achat, num_facture
FROM utilisateur
RIGHT JOIN commande ON utilisateur.id = commande.utilisateur_id
```

Résultats :

id	prenom	nom	utilisateur_id	date_achat	num_facture
1	Aimée	Marechal	1	2013-01-23	A00103
1	Aimée	Marechal	1	2013-02-14	A00104
2	Esmée	Lefort	2	2013-02-17	A00105
3	Marine	Prevost	3	2013-02-21	A00106
NULL	NULL	NULL	5	2013-03-02	A00107

Ce résultat montre que la facture A00107 est liée à l'utilisateur numéro 5. Or, cet utilisateur n'existe pas ou n'existe plus. Grâce à RIGHT JOIN, l'achat est tout de même affiché mais les informations liées à l'utilisateur sont remplacé par NULL.

SQL FULL JOIN

Dans le langage SQL, la commande FULL JOIN (ou FULL OUTER JOIN) permet de faire une jointure entre 2 tables. L'utilisation de cette commande permet de combiner les résultats des 2 tables, les associer entre eux grâce à une condition et remplir avec des valeurs NULL si la condition n'est pas respectée.

Syntaxe

Pour retourner les enregistrements de table1 et table2, il convient d'utiliser une requête SQL avec une syntaxe telle que celle-ci :

```
SELECT *
FROM table1
FULL JOIN table2 ON table1.id = table2.fk_id
```

Cette requête peut aussi être conçu de cette façon :

```
SELECT *
FROM table1
FULL OUTER JOIN table2 ON table1.id = table2.fk_id
```

La condition présentée ici consiste à lier les tables sur un identifiant, mais la condition peut être définie sur d'autres champs.

Exemple

Prenons l'exemple d'une base de données qui contient une table utilisateur ainsi qu'une table commande qui contient toutes les ventes.

Table utilisateur :

id	prenom	nom	email	ville	actif
1	Aimée	Marechal	aime.marechal@example.com	Paris	1
2	Esmée	Lefort	esmee.lefort@example.com	Lyon	0
3	Marine	Prevost	m.prevost@example.com	Lille	1
4	Luc	Rolland	lucrolland@example.com	Marseille	1

Table commande :

utilisateur_id	date_achat	num_facture	prix_total
1	2013-01-23	A00103	203.14
1	2013-02-14	A00104	124.00
2	2013-02-17	A00105	149.45
3	2013-02-21	A00106	235.35
5	2013-03-02	A00107	47.58

Il est possible d'utiliser FULL JOIN pour lister tous les utilisateurs ayant effectué ou non une vente, et de lister toutes les ventes qui sont associées ou non à un utilisateur. La requête SQL est la suivante :

```
SELECT id, prenom, nom, utilisateur_id, date_achat, num_facture
FROM utilisateur
FULL JOIN commande ON utilisateur.id = commande.utilisateur_id
```

Résultat :

id	prenom	nom	utilisateur_id	date_achat	num_facture
1	Aimée	Marechal	1	2013-01-23	A00103
1	Aimée	Marechal	1	2013-02-14	A00104
2	Esmée	Lefort	2	2013-02-17	A00105
3	Marine	Prevost	3	2013-02-21	A00106
4	Luc	Rolland	NULL	NULL	NULL
NULL	NULL		5	2013-03-02	A00107

Ce résultat affiche bien l'utilisateur numéro 4 qui n'a effectué aucun achat. Le résultat retourne également la facture A00107 qui est associée à un utilisateur qui n'existe pas (ou qui n'existe plus). Dans les cas où il n'y a pas de correspondance avec l'autre table, les valeurs des colonnes valent NULL.

SQL SELF JOIN

En SQL, un SELF JOIN correspond à une jointure d'une table avec elle-même. Ce type de requête n'est pas si commun mais très pratique dans le cas où une table lie des informations avec des enregistrements de la même table.

Syntaxe

Pour effectuer un SELF JOIN, la syntaxe de la requête SQL est la suivante :

```
SELECT `t1`.`nom_colonne1`, `t1`.`nom_colonne2`, `t2`.`nom_colonne1`,
`t2`.`nom_colonne2`
FROM `table` as `t1`
LEFT OUTER JOIN `table` as `t2` ON `t2`.`fk_id` = `t1`.`id`
```

Ici la jointure est effectuée avec un LEFT JOIN, mais il est aussi possible de l'effectuer avec d'autres types de jointures.

Exemple

Un exemple potentiel pourrait être une application d'un intranet d'entreprise qui possède la table des employés avec la hiérarchie entre eux. Les employés peuvent être dirigé par un supérieur direct qui se trouve lui-même dans la table.

Table utilisateur :

id	prenom	nom	email	manager_id
1	Sebastien	Martin	s.martin@example.com	NULL
2	Gustave	Dubois	g.dubois@example.com	NULL
3	Georgette	Leroy	g.leroy@example.com	1
4	Gregory	Roux	g.roux@example.com	2

Les enregistrements de la table ci-dessus montre bien des employés. Les premiers employés n'ont pas de supérieur, tandis que les employés n°3 et n°4 ont respectivement pour supérieur l'employé n°1 et l'employé n°2.

Il est possible de lister sur une même ligne les employés avec leurs supérieurs direct, grâce à une requête telle que celle-ci :

```
SELECT `u1`.`u_id`, `u1`.`u_nom`, `u2`.`u_id`, `u2`.`u_nom`
FROM `utilisateur` as `u1`
LEFT OUTER JOIN `utilisateur` as `u2` ON `u2`.`u_manager_id` = `u1`.`u_id`
```

Résultat :

u1_id	u1_prenom	u1_nom	u1_email	u1_manager_id	u2_prenom	u2_nom
1	Sebastien	Martin	s.martin@example.com	NULL	NULL	NULL
2	Gustave	Dubois	g.dubois@example.com	NULL	NULL	NULL
3	Georgette	Leroy	g.leroy@example.com	1	Sebastien	Martin
4	Gregory	Roux	g.roux@example.com	2	Gustave	Dubois

SQL NATURAL JOIN

Dans le langage SQL, la commande NATURAL JOIN permet de faire une jointure naturelle entre 2 tables. Cette jointure s'effectue à la condition qu'il y ai des colonnes du même nom et de même type dans les 2 tables. Le résultat d'une jointure naturelle est la création d'un tableau avec autant de lignes qu'il y a de paires correspondant à l'association des colonnes de même nom.

A noter : puisqu'il faut le même nom de colonne sur les 2 tables, cela empêche d'utiliser certaines règles de nommages pour le nom des colonnes. Il n'est par exemple pas possible de préfixer le nom des colonnes sous peine d'avoir malheureusement 2 nom de colonnes différents.

Syntaxe

La jointure naturelle de 2 tables peut s'effectuer facilement, comme le montre la requête SQL suivante :

```
SELECT *
FROM table1
NATURAL JOIN table2
```

L'avantage d'un NATURAL JOIN c'est qu'il n'y a pas besoin d'utiliser la clause ON.

Exemple

Une utilisation classique d'une telle jointure pourrait être l'utilisation dans une application qui utilise une table utilisateur et une table pays. Si la table utilisateur contient une colonne pour l'identifiant du pays, il sera possible d'effectuer une jointure naturelle.

Table « utilisateur » :

user_id	user_prenom	user_ville	pays_id
1	Jérémie	Paris	1
2	Damien	Lyon	2
3	Sophie	Marseille	NULL
4	Yann	Lille	9999
5	Léa	Paris	1

Table « pays » :

pays_id	pays_nom
1	France
2	Canada
3	Belgique
4	Suisse

Pour avoir la liste de tous les utilisateurs avec le pays correspondant, il est possible d'effectuer une

requête SQL similaire à celle-ci :

```
SELECT *
FROM utilisateur
NATURAL JOIN pays
```

Cette requête retournera le résultat suivant :

pays_id	user_id	user_prenom	user_ville	pays_nom
1	1	Jérémie	Paris	France
2	2	Damien	Lyon	Canada
NULL	3	Sophie	Marseille	NULL
9999	4	Yann	Lille	NULL
1	5	Léa	Paris	France

Cet exemple montre qu’il y a bien eu une jointure entre les 2 tables grâce à la colonne « pays_id » qui se trouve dans l’une et l’autre des tables.

SQL Sous-requête

Dans le langage SQL une sous-requête (aussi appelé « requête imbriquée » ou « requête en cascade ») consiste à exécuter une requête à l'intérieur d'une autre requête. Une requête imbriquée est souvent utilisée au sein d'une clause WHERE ou de HAVING pou remplacer une ou plusieurs constante.

Syntaxe

Il y a plusieurs façons d'utiliser les sous-requêtes. De cette façon il y a plusieurs syntaxes envisageables pour utiliser des requêtes dans des requêtes.

Requête imbriquée qui retourne un seul résultat

L'exemple ci-dessous est une exemple typique d'une sous-requête qui retourne un seul résultat à la requête principale.

```
SELECT *
FROM `table`
WHERE `nom_colonne` = (
    SELECT `valeur`
    FROM `table2`
    LIMIT 1
  )
```

Cet exemple montre une requête interne (celle sur « table2″) qui renvoi une seule valeur. La requête externe quant à elle, va chercher les résultat de « table » et filtre les résultats à partir de la valeur retournée par la requête interne.

A noter : il est possible d'utiliser n'importe quel opérateur d'égalité tel que =, >, <, >=, <= ou <>.

Requête imbriquée qui retourne une colonne

Une requête imbriquée peut également retournée une colonne entière. Dès lors, la requête externe peut utiliser la commande IN pour filtrer les lignes qui possèdent une des valeurs retournées par la requête interne. L'exemple ci-dessous met en évidence un tel cas de figure :

```
SELECT *
FROM `table`
WHERE `nom_colonne` IN (
    SELECT `colonne`
    FROM `table2`
    WHERE `cle_etrangere` = 36
  )
```

Exemple

La suite de cet article présente des exemples concrets utilisant les sous-requêtes.

Imaginons un site web qui permet de poser des questions et d’y répondre. Un tel site possède une base de données avec une table pour les questions et une autre pour les réponses.

Table « question » :

q_id	q_date_ajout	q_titre	q_contenu
1	2013-03-24 12:54:32	Comment réparer un ordinateur?	Bonjour, j'ai mon ordinateur de cassé, comment puis-je procéder pour le réparer?
2	2013-03-26 19:27:41	Comment changer un pneu?	Quel est la meilleur méthode pour changer un pneu facilement ?
3	2013-04-18 20:09:56	Que faire si un appareil est cassé?	Est-il préférable de réparer les appareils électriques ou d'en acheter de nouveaux?
4	2013-04-22 17:14:27	Comment faire nettoyer un clavier d'ordinateur?	Bonjour, sous mon clavier d'ordinateur il y a beaucoup de poussière, comment faut-il procéder pour le nettoyer? Merci.

Table « reponse » :

r_id	r_fk_question_id	r_date_ajout	r_contenu
1	1	2013-03-27 07:44:32	Bonjour. Pouvez-vous expliquer ce qui ne fonctionne pas avec votre ordinateur? Merci.
2	1	2013-03-28 19:27:11	Bonsoir, le plus simple consiste à faire appel à un professionnel pour réparer un ordinateur. Cordialement,
3	2	2013-05-09 22:10:09	Des conseils son disponible sur internet sur ce sujet.
4	3	2013-05-24 09:47:12	Bonjour. Ça dépend de vous, de votre budget et de vos préférence vis-à-vis de l'écologie. Cordialement,

Requête imbriquée qui retourne un seul résultat

Avec une telle application, il est peut-être utile de connaître la question liée à la dernière réponse ajoutée sur l’application. Cela peut être effectué via la requête SQL suivante :

```
SELECT *
FROM `question`
WHERE q_id = (
    SELECT r_fk_question_id
    FROM `reponse`
    ORDER BY r_date_ajout DESC
    LIMIT 1
  )
```

Une telle requête va retourner la ligne suivante :

q_id	q_date_ajout	q_titre	q_contenu
3	2013-04-18 20:09:56	Que faire si un appareil est cassé?	Est-il préférable de réparer les appareils électriques ou d'en acheter de nouveaux?

Ce résultat démontre que la question liée à la dernière réponse sur le forum est bien trouvée à partir de ce résultat.

Requête imbriquée qui retourne une colonne

Imaginons maintenant que l'ont souhaite obtenir les questions liées à toutes les réponses comprises entre 2 dates. Ces questions peuvent être récupérée par la requête SQL suivante :

```
SELECT *
FROM `question`
WHERE q_id IN (
    SELECT r_fk_question_id
    FROM `reponse`
    WHERE r_date_ajout BETWEEN '2013-01-01' AND '2013-12-31'
  )
```

Résultats :

q_id	q_date_ajout	q_titre	q_contenu
1	2013-03-24 12:54:32	Comment réparer un ordinateur?	Bonjour, j'ai mon ordinateur de cassé, comment puis-je procéder pour le réparer?
2	2013-03-26 19:27:41	Comment changer un pneu?	Quel est la meilleur méthode pour changer un pneu facilement ?
3	2013-04-18 20:09:56	Que faire si un appareil est cassé?	Est-il préférable de réparer les appareils électriques ou d'en acheter de nouveaux?

Une telle requête permet donc de récupérer les questions qui ont eu des réponses entre 2 dates. C'est pratique dans notre cas pour éviter d'obtenir des réponses qui n'ont pas eu de réponses du tout ou pas de nouvelles réponses depuis longtemps.

SQL EXISTS

Dans le langage SQL, la commande EXISTS s'utilise dans une clause conditionnelle pour savoir s'il y a une présence ou non de lignes lors de l'utilisation d'une sous-requête.

A noter : cette commande n'est pas à confondre avec la clause IN. La commande EXISTS vérifie si la sous-requête retourne un résultat ou non, tandis que IN vérifie la concordance d'une à plusieurs données.

Syntaxe

L'utilisation basique de la commande EXISTS consiste à vérifier si une sous-requête retourne un résultat ou non, en utilisant EXISTS dans la clause conditionnelle. La requête externe s'exécutera uniquement si la requête interne retourne au moins un résultat.

```
SELECT nom_colonne1
FROM `table1`
WHERE EXISTS (
    SELECT nom_colonne2
    FROM `table2`
    WHERE nom_colonne3 = 10
  )
```

Dans l'exemple ci-dessus, s'il y a au moins une ligne dans table2 dont nom_colonne3 contient la valeur 10, alors la sous-requête retournera au moins un résultat. Dès lors, la condition sera vérifiée et la requête principale retournera les résultats de la colonne nom_colonne1 de table1.

Exemple

Dans le but de montrer un exemple concret d'application, imaginons un système composé d'une table qui contient des commandes et d'une table contenant des produits.

Table commande :

c_id	c_date_achat	c_produit_id	c_quantite_produit
1	2014-01-08	2	1
2	2014-01-24	3	2
3	2014-02-14	8	1
4	2014-03-23	10	1

Table produit :

p_id	p_nom	p_date_ajout	p_prix
2	Ordinateur	2013-11-17	799.9
3	Clavier	2013-11-27	49.9
4	Souris	2013-12-04	15
5	Ecran	2013-12-15	250

Il est possible d’effectuer une requête SQL qui affiche les commandes pour lesquels il y a effectivement un produit. Cette requête peut être interprétée de la façon suivante :

```
SELECT *
FROM commande
WHERE EXISTS (
    SELECT *
    FROM produit
    WHERE c_produit_id = p_id
)
```

Résultat :

c_id	c_date_achat	c_produit_id	c_quantite_produit
1	2014-01-08	2	1
2	2014-01-24	3	2

Le résultat démontre bien que seul les commandes n°1 et n°2 ont un produit qui se trouve dans la table produit (cf. la condition c_produit_id = p_id). Cette requête est intéressante sachant qu’elle n’influence pas le résultat de la requête principale, contrairement à l’utilisation d’une jointure qui va concaténer les colonnes des 2 tables jointes.

SQL ALL

Dans le langage SQL, la commande ALL permet de comparer une valeur dans l'ensemble de valeurs d'une sous-requête. En d'autres mots, cette commande permet de s'assurer qu'une condition est « égale », « différente », « supérieure », « inférieure », « supérieure ou égale » ou « inférieure ou égale » pour tous les résultats retourné par une sous-requête.

Syntaxe

Cette commande s'utilise dans une clause conditionnelle entre l'opérateur de condition et la sous-requête. L'exemple ci-dessous montre un exemple basique :

```
SELECT *
FROM table1
WHERE condition > ALL (
    SELECT *
    FROM table2
    WHERE condition2
)
```

A savoir : les opérateur conditionnels peuvent être les suivants : =, <, >, <>, !=, <=, >=, !> ou !<.

Exemple

Imaginons une requête similaire à la syntaxe de base présentée précédemment :

```
SELECT colonne1
FROM table1
WHERE colonne1 > ALL (
    SELECT colonne1
    FROM table2
)
```

Avec cette requête, si nous supposons que dans table1 il y a un résultat avec la valeur 10, voici les différents résultats de la conditions selon le contenu de table2 :

- La condition est vrai (cf. TRUE) si table2 contient {-5,0,+5} car toutes les valeurs sont inférieure à 10
- La condition est fausse (cf. FALSE) si table2 contient {12,6,NULL,-100} car au moins une valeur est inférieure à 10
- La condition est non connue (cf. UNKNOW) si table2 est vide

SQL ANY / SOME

Dans le langage SQL, la commande ANY (ou SOME) permet de comparer une valeur avec le résultat d'une sous-requête. Il est ainsi possible de vérifier si une valeur est « égale », « différente », « supérieur », « supérieur ou égale », « inférieur » ou « inférieur ou égale » pour au moins une des valeurs de la sous-requête.

A noter : le mot-clé SOME est un alias de ANY, l'un et l'autre des termes peut être utilisé.

Syntaxe

Cette commande s'utilise dans une clause conditionnelle juste après un opérateur conditionnel et juste avant une sous-requête. L'exemple ci-dessous démontre une utilisation basique de ANY dans une requête SQL :

```
SELECT *
FROM table1
WHERE condition > ANY (
    SELECT *
    FROM table2
    WHERE condition2
)
```

Cette requête peut se traduire de la façon suivante : sélectionner toutes les colonnes de table1, où la condition est supérieure à n'importe quel résultat de la sous-requête.

A savoir : les opérateur conditionnels peuvent être les suivants : =, <, >, <>, !=,<=, >=, !> ou !<.

Exemple

En se basant sur l'exemple relativement simple présenté ci-dessus, il est possible d'effectuer une requête concrète qui utilise la commande ANY :

```
SELECT colonne1
FROM table1
WHERE colonne1 > ANY (
    SELECT colonne1
    FROM table2
)
```

Supposons que la table1 possède un seul résultat dans lequel colonne1 est égal à 10.

- La condition est vrai (cf. TRUE) si table2 contient {21,14,7} car il y a au moins une valeur inférieure à 10
- La condition est fausse (cf. FALSE) si table2 contient {20,10} car aucune valeur est strictement inférieure à 10
- La condition est non connue (cf. UNKNOW) si table2 est vide

Astuce

La commande IN est équivalent à l’opérateur = suivi de ANY.

Index SQL

En SQL, les index sont des ressources très utiles qui permettent d'accéder plus rapidement aux données. Cette page explique le fonctionnement des index et leurs intérêts pour accroître les performances de lectures des données.

Analogie pour comprendre les index en SQL

Un index, dans le domaine bibliographique, permet de lister les mots-clés importants abordés dans un ouvrage et d'indiquer les pages où le mot est mentionné. Ainsi, un lecteur qui recherche une thématique spécifique peut se baser sur cet index pour trouver les pages qui abordent le sujet. Ainsi un index est une ressource non indispensable, mais c'est un un gain de temps terrible pour l'utilisateur qui accède facilement à l'information recherchée.

Index en SQL

Un index, dans une base de données se base sur le même principe qu'un index dans un livre. Avec un index placé sur une ou plusieurs colonnes le système d'une base de données peut rechercher les données d'abord sur l'index et s'il trouve ce qu'il cherche il saura plus rapidement où se trouve les enregistrements concernés.

Ces petites ressources ont toutefois leurs inconvénients car cela occupe de l'espace supplémentaire dans la base de données. Par ailleurs, l'insertion de données est plus long car les index sont mis à jour à chaque fois que des données sont insérées.

Généralement un index pourra être utilisé dans les requêtes utilisant les clauses WHERE, GROUP BY ou ORDER BY. Lorsqu'une base de données possède un grand nombre d'enregistrements (exemple: plusieurs milliers ou plusieurs millions de lignes) un index permet de gagner un temps précieux pour la lecture de données.

SQL CREATE INDEX

En SQL, la commande CREATE INDEX permet de créer un index. L'index est utile pour accélérer l'exécution d'une requête SQL qui lit des données et ainsi améliorer les performances d'une application utilisant une base de données.

Syntaxe

Créer un index ordinaire

La syntaxe basique pour créer un index est la suivante :

```
CREATE INDEX `index_nom` ON `table`;
```

Il est également possible de créer un index sur une seule colonne en précisant la colonne sur laquelle doit s'appliquer l'index :

```
CREATE INDEX `index_nom` ON `table` (`colonne1`);
```

L'exemple ci-dessus va donc insérer l'index intitulé « index_nom » sur la table nommée « table » uniquement sur la colonne « colonne1″. Pour insérer un index sur plusieurs colonnes il est possible d'utiliser la syntaxe suivante :

```
CREATE INDEX `index_nom` ON `table` (`colonne1`, `colonne2`);
```

L'exemple ci-dessus permet d'insérer un index les 2 colonnes : colonne1 et colonne2.

Créer un index unique

Un index unique permet de spécifier qu'une ou plusieurs colonnes doivent contenir des valeurs uniques à chaque enregistrement. Le système de base de données retournera une erreur si une requête tente d'insérer des données qui feront doublons sur la clé d'unicité. Pour insérer un tel index il suffit d'exécuter une requête SQL respectant la syntaxe suivante :

```
CREATE UNIQUE INDEX `index_nom` ON `table` (`colonne1`);
```

Dans cet exemple un index unique sera créé sur la colonne nommée colonne1. Cela signifie qu'il ne peut pas y avoir plusieurs fois la même valeur sur 2 enregistrements distincts contenus dans cette table.

Il est également possible de créer un index d'unicité sur 2 colonnes, en respectant la syntaxe suivante :

```
CREATE UNIQUE INDEX `index_nom` ON `table` (`colonne1`, `colonne2`);
```

Convention de nommage

Il n'existe pas de convention de nommage spécifique sur le nom des index, juste des suggestions de quelques développeurs et administrateurs de bases de données. Voici une liste de suggestions de préfixes à utiliser pour nommer un index :

- Préfixe « **PK_** » pour **P**rimary **K**ey (traduction : clé primaire)
- Préfixe « **FK_** » pour **F**oreign **K**ey (traduction : clé étrangère)
- Préfixe « **UK_** » pour **U**nique **K**ey (traduction : clé unique)
- Préfixe « **UX_** » pour **U**nique **I**ndex (traduction : index unique)
- Préfixe « **IX_** » pour chaque autre **I**nde**X**

SQL EXPLAIN

Dans le langage SQL, l'instruction EXPLAIN est à utiliser juste avant un SELECT et permet d'afficher le plan d'exécution d'une requête SQL. Cela permet de savoir de quelle manière le Système de Gestion de Base de Données (SGBD) va exécuter la requête et s'il va utiliser des index et lesquels.

En utilisant cette commande la requête ne renverra pas les résultats du SELECT mais plutôt une analyse de cette requête.

A noter : le résultat de cette instruction est différent selon les SGBD, tel que MySQL ou PostgreSQL. Par ailleurs, le nom de cette instruction diffère pour certains SGBD :

- **MySQL :** EXPLAIN
- **PostgreSQL :** EXPLAIN
- **Oracle :** EXPLAIN PLAN
- **SQLite :** EXPLAIN QUERY PLAN
- **SQL Server :**
  SET SHOWPLAN_ALL : informations estimées d'une requête SQL, affiché au format textuel détaillé
  SET SHOWPLAN_TEXT : informations estimées d'une requête SQL, affiché au format textuel simple
  SET SHOWPLAN_XML : informations estimées d'une requête SQL, affiché au format XML
  SET STATISTICS PROFILE : statistiques sur l'exécution d'une requête SQL, affiché au format textuel
  SET STATISTICS XML : statistiques sur l'exécution d'une requête SQL, affiché au format XML
- **Firebird :** SET PLANONLY ON; puis l'exécution de la requête SQL à analyser

Syntaxe

La syntaxe ci-dessous représente une requête SQL utilisant la commande EXPLAIN pour MySQL ou PostgreSQL :

```
EXPLAIN SELECT *
FROM `user`
ORDER BY `id` DESC
```

Rappel : dans cet exemple, la requête retournera des informations sur le plan d'exécution, mais n'affichera pas les « vrai » résultats de la requête.

Exemple

Pour expliquer concrètement le fonctionnement de l'instruction EXPLAIN nous allons prendre une table des fuseaux horaires en PHP. Cette table peut être créé à partir de la requête SQL suivante :

```
CREATE TABLE IF NOT EXISTS `timezones` (
 `timezone_id` int(10) unsigned NOT NULL AUTO_INCREMENT,
 `timezone_groupe_fr` varchar(50) DEFAULT NULL,
 `timezone_groupe_en` varchar(50) DEFAULT NULL,
 `timezone_detail` varchar(100) DEFAULT NULL,
 PRIMARY KEY (`timezone_id`)
```

```
) ENGINE=InnoDB DEFAULT CHARSET=utf8 AUTO_INCREMENT=698;
```

La requête ci-dessous permet de mieux comprendre la structure et les index de cette table.

Imaginons que l'ont souhaite compter le nombre de fuseaux horaires par groupe, pour cela il est possible d'utiliser la requête SQL suivante :

```
SELECT timezone_groupe_fr, COUNT(timezone_detail) AS total_timezone
FROM `timezones`
GROUP BY timezone_groupe_fr
ORDER BY timezone_groupe_fr ASC
```

Analyse de la requête SQL

Nous allons voir dans notre exemple comment MySQL va exécuter cette requête. Pour cela, il faut utiliser l'instruction EXPLAIN :

```
EXPLAIN SELECT timezone_groupe_fr, COUNT(timezone_detail) AS total_timezone
FROM `timezones`
GROUP BY timezone_groupe_fr
ORDER BY timezone_groupe_fr ASC
```

Le retour de cette requête SQL est le suivant :

```
EXPLAIN SELECT timezone_groupe_fr, COUNT( timezone_detail ) AS total_timezone
FROM `timezones`
GROUP BY timezone_groupe_fr
ORDER BY timezone_groupe_fr ASC
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	timezones	ALL	NULL	NULL	NULL	NULL	421	Using temporary; Using filesort

Requête SQL avec EXPLAIN sans index

Dans cet exemple on constate les champs suivants :

- **id :** identifiant de SELECT
- **select_type :** type de cause SELECT (exemple : SIMPLE, PRIMARY, UNION, DEPENDENT UNION, SUBQUERY, DEPENDENT SUBSELECT ou DERIVED)
- **table :** table à laquelle la ligne fait référence
- **type :** le type de jointure utilisé (exemple : system, const, eq_ref, ref, ref_or_null, index_merge, unique_subquery, index_subquery, range, index ou ALL)
- **possible_keys :** liste des index que MySQL pourrait utiliser pour accélérer l'exécution de la requête. Dans notre exemple, aucun index n'est disponible pour accélérer l'exécution de la requête SQL
- **key :** cette colonne présente les index que MySQL a décidé d'utiliser pour l'exécution de la requête
- **key_len :** indique la taille de la clé qui sera utilisée. S'il n'y a pas de clé, cette colonne renvois NULL
- **ref :** indique quel colonne (ou constante) sont utilisés avec les lignes de la table
- **rows :** estimation du nombre de ligne que MySQL va devoir analyser examiner pour exécuter la requête
- **Extra :** information additionnelle sur la façon dont MySQL va résoudre la requête. Si cette colonne retourne des résultats, c'est qu'il y a potentiellement des index à utiliser pour

optimiser les performances de la requête SQL. Le message « using temporary » permet de savoir que MySQL va devoir créer une table temporaire pour exécuter la requête. Le message « using filesort » indique quant à lui que MySQL va devoir faire un autre passage pour retourner les lignes dans le bon ordre

Ajout d’un index

Il est possible d’ajouter un index sur la colonne « timezone_groupe_fr » à la table qui n’en avait pas.

```
ALTER TABLE `timezones` ADD INDEX ( `timezone_groupe_fr` );
```

L’ajout de cet index va changer la façon dont MySQL peut exécuter une requête SQL. En effectuant la même requête que tout à l’heure, les résultats seront différent.

```
EXPLAIN SELECT timezone_groupe_fr, COUNT( timezone_detail ) AS total_timezone
FROM `timezones`
GROUP BY timezone_groupe_fr
ORDER BY timezone_groupe_fr ASC
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	timezones	index	*NULL*	index_timezone_groupe_fr	153	*NULL*	471	

Requête SQL avec EXPLAIN avec index

Dans ce résultat il est possible de constater que MySQL va utiliser un l’index « index_timezone_groupe_fr » et qu’il n’y a plus aucune information complémentaire d’indiquée dans la colonne « Extra ».

Commentaires en SQL

Il peut être intéressant d'insérer des commentaires dans les requêtes SQL pour mieux s'y retrouver lorsqu'il y a de grosses requêtes complexes. Il y a plusieurs façon de faire des commentaires dans le langage SQL, qui dépendent notamment du Système de Gestion de Base de Données utilisées (SGBD) et de sa version.

Commentaire double tiret : –

Le double tiret permet de faire un commentaire jusqu'à la fin de la ligne.

Exemple

```
SELECT *    -- tout sélectionner
FROM table1 -- dans la table "table1"
```

Compatibilité

- Depuis la version 3.23.3 de MySQL
- PostgreSQL
- Oracle
- SQLite

Commentaire dièse :

Le symbole dièse permet de faire un commentaire jusqu'à la fin de la ligne.

Exemple

```
SELECT *    # tout sélectionner
FROM table1 # dans la table "table1"
```

Compatibilité

- MySQL

Commentaire multi-ligne : /* et */

Le commentaire multi-ligne à l'avantage de pouvoir indiquer où commence et où se termine le commentaire. Il est donc possible de l'utiliser en plein milieu d'une requête SQL sans problème.

Exemple

```
SELECT *    /* tout sélectionner */
FROM table1 /* dans la table "table1" */
WHERE 1 = /* exemple en milieu de requete */ 1
```

Compatibilité

- MySQL
- PostgreSQL
- Oracle

- SQL Server
- SQLite

Bug potentiel pour les commentaires sur une ligne

Attention, dans certains contextes, si vous utilisez un système qui va supprimer les retours à la ligne, votre requête sera uniquement sur une ligne. Dans une telle situation, un commentaire effectué avec « – » dans une requête peut donc créer un bug.

Requête SQL sur plusieurs lignes :

```
SELECT *    -- tout sélectionner
FROM table1 -- dans la table "table1"
```

Même requête SQL sur une ligne :

```
SELECT * -- tout sélectionner FROM table1 -- dans la table "table1"
```

En conséquent il faut se méfier de librairies externes qui peuvent ré-écrire les requêtes SQL ou alors tout simplement se méfier de copier/coller.